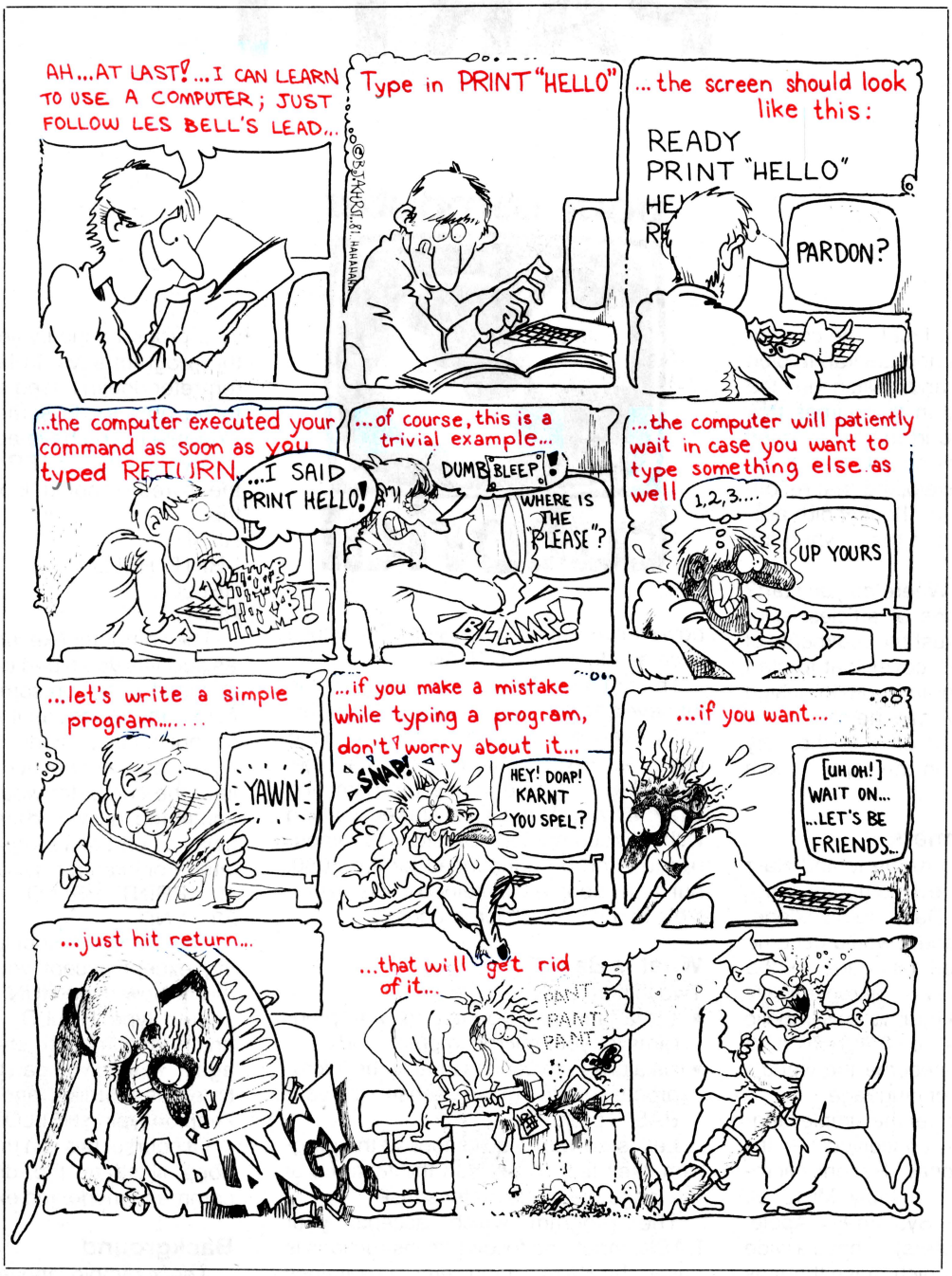
BONUS offer
BASICally free!

Basic for Birdwatchers

How to learn BASIC in only 16 easy lessons (eight this month and eight next month), in the comfort and privacy of your own home.



If you've never used a computer before, and if you're unsure where to start, Les Bell can teach you the BASICs of programming in this series. First published in ***Your Computer*** *in instalments from the September/October 1981 issue to the April 1983 issue, the series has been on constant back-order ever since. So here it is in full — and as relevant as ever.*

PART I

WHEN YOU buy your first home computer, and take it home and unwrap it, plug everything together and switch on, the temptation is to load in the games that were supplied and spend some time trying them out. But once the novelty wears out, you're set to start developing that record cataloguing system or household accounting package — and that's when the horror starts.

How do you do it? Where do you start? That's what this series of articles is intended to teach, not just the rudiments of BASIC, although we'll cover that to start with, but how to program; how to design programs and how to build them.

We're going to start off with the very basics (if you'll pardon the pun) — with BASIC itself.

Not Just for Beginners

In the sixties when Kemeny and Kurz were developing a new computer language for teaching at Dartmouth College, neither of them realised it would become the most widely-used language in the world. The deviously acronymically named Beginners' All-purpose Symbolic Instruction Code has now left behind its humble origins, and become the world's most popular computer language.

But BASIC is no longer the simple language its originators conceived — it's grown teeth, sharp ones. Recent microcomputer BASICs — notably Microsoft (used in the TRS-80, System 80, Apple, PET and other computers) — have a wide range of extensions which make them as powerful as earlier 'big system' languages such as Algol or FORTRAN.

The big thing BASIC has going for it is that everybody has it on their system. Virtually every personal computer sold these days has BASIC built into the machine. Just plug it in, and you're programming straight away. When I first started working with microcomputers, BASIC was rumoured to soon be announced for the 8080, but many people maintained it was impossible to run it on a microprocessor!

What is Basic?

Two things really:

- it is a language you use to write programs for your microcomputer, and
- it is a program used to enable the microprocessor to make sense of your BASIC program.

Let' start with a quick look at the program itself first of all; then the features of the language make better sense.

The program which accepts your BASIC input and follows its instructions is called the BASIC interpreter. It examines your program line by line, and does what the program says. In fact, the BASIC interpreter doesn't need a program in order to be able to do something useful. For example (if you have access to a computer try this), type in PRINT "HELLO" and the screen should look something like this:

```
READY
PRINT "HELLO"
HELLO
READY
```

The computer executed your command as soon as you typed carriage return (this is called 'enter' on some machines.) This command mode, as it is called, is useful for quick calculations. But if you were to type a lone number at the beginning of the line, the computer wouldn't execute your command, but would instead store the line away as part of a program to be executed later. For example, type:

```
10  PRINT "HELLO"
20  END
```

The computer should have done nothing except accept your lines and store them. Now, type RUN, and the computer should type HELLO. Type RUN again, and the computer should type HELLO again. In fact, you can type RUN as many times as you like, and the computer will keep on typing HELLO.

Of course, this is a trivial example, but it does illustrate the difference between command mode and execute mode.

Background

The example above will work on just

Fig 1

about any computer, particularly the TRS-80 and similar personal computers. But there are different versions of BASIC, and so programs written in one dialect of the language will need to be translated or re-written to run on a computer with a different version.

The most common dialect in the microcomputer world is Microsoft BASIC (also known as MBASIC). This is the BASIC used in the TRS-80, the PET and other popular personal computers. But there are others, and it's as well to know about them, in case you come across one which behaves in a strange manner.

For example CBASIC is a compiler, rather than an interpreter like MBASIC. This means that there is no command mode, and you can only write programs. You do this by typing your program into a text file stored on floppy disk. Then by typing CBAS2 YOURPRO you instruct the CBASIC compiler to read through your program and create an intermediate file, which is what the computer will actually use as its instructions when the program is run.

You then type CRUN2 YOURPROG and the CBASIC run-time package will load and execute the intermediate file. As you can see, it's rather more complex than using a BASIC interpreter.

There are also different BASIC interpreters. Another popular one, for example, is NorthStar's; it's similar to MBASIC, but has some differences in string handling which can sometimes throw the beginner.

There is a standard version called ANSI BASIC (American National Standards Institution), but it really only covers the basics, if you'll excuse the pun. Almost every personal computer BASIC exceeds the standard, and that is where the incompatibilities lie. For example, some BASICs have matrix arithmetic functions, and programs using these can be quite tricky to translate into a language that doesn't have them.

Some extensions can't really be said to be part of the language. A few computers have a TIME or TIME$ function, for example, which reads the computer's internal clock. If your computer doesn't have a clock and a similar function, you may find it difficult to translate a program to run on your computer.

But the biggest area of difficulty is the different types of graphics used by different computers. For example, the TRS-80 uses SET and RESET statements to light up individual points on the screen, while the Apple has HLIN and VLIN statements which enable it to draw a line in one go. Translating Apple graphics programs to run on the TRS-80 is very difficult indeed, even disregarding the incompatibilities in their hardware.

Because of these difficulties, you may find that some of the examples in this series will not run on your computer. However, wherever possible, where differences exist between different computers we will point them out.

Getting Down to It

If you've got a computer, a piece of paper and a pencil, that's all you'll need to work through the examples we'll be giving. You will find the journey enlightening and interesting.

Computers are really just like glorified adding machines. You type in some information, the computer crunches it around in various ways, and gives you back your answer, either through the printer or on the display. We'll generally use the display.

The keyboard on your computer is similar to an electric typewriter, but there are a few new keys.

At the right end, you should see a key (perhaps bigger than the others or coloured differently) marked CARRIAGE RETURN or RETURN or CR. On some computers it's called ENTER. This is the key you hit at the end of each line. It tells the computer that you've finished typing a line, and that you want it to take a look at what you've typed and do what you've asked. For example, when you want to run a program, it's no use just typing RUN, because the computer will patiently wait in case you want to type something else as well. You've got to hit the RETURN key to make it do something.

If you're typing in a program, then each line will start with a line number, and the computer will automatically put each line in memory, and keep the lines in numerical order. That's why we generally type in lines by tens: 10, 20, 30 and so on. This way, if we want to add a line between lines 10 and 20, we just number it 15, and it will go in the right place.

If you make a mistake while typing a program line, don't worry about it — just hit RETURN and retype the line (with its line number, of course).

The new version of the line will overwrite the old one. If you want to delete a line, just type its line number and hit RETURN. That will get rid of it.

Our first job is to get information in and out of the computer. After all, it's easy to write programs that don't do any input or output, but then you can't tell if they're doing anything! For example, type in this program

```
10 GOTO 10
```

and then type RUN followed by RETURN. What's your computer doing? Not much of anything useful, that's for sure. To stop it, look for the BREAK key and press it. If there's no break key, then find the 'control' key, and hold it down while you press the C key. That should stop it. Make sure you know the standard way to stop a program

while it's running — usually BREAK or control-C.

Now let's get on and do something useful.

To get rid of that useless program, type 'NEW' followed by RETURN. That tells the computer that you're about to start typing in a new program. On some computers, the command is SCR or SCRATCH, to get rid of the old program.

As we've seen, whenever you start a line with a number, the computer just stores that line away in sequences, ready for when you want to run a program..Lets write a simple program. Type this in:

```
New
10  PRINT   "YOUR COMPUTER ";
20  GOTO 10
```

Don't forget the semicolon at the end of line 10 or the space before the end of the quote marks — it's very important. If you make a mistake, don't worry — just retype the entire line. Later we'll see how to fix errors without wearing our fingers out.

To take a look at your program, type LIST, and the computer will type it all out. It doesn't matter in which order you typed the lines in, the computer will always list them out in numerical order.

Before we analyse how it works, let's see what the program does. Type RUN.

And don't forget the all important carriage return.

Well, what did you get? It should look something like Fig. 1, allowing for the fact that your screen is probably of different dimensions from mine. If it's still going, then stop it by hitting BREAK or control-C keys.

How does it work? Look at line 10. It tells the computer to print everything after the word PRINT. Then there's the words YOUR COMPUTER (and a space), but they're between quote marks.

These tell the computer that everything between the quotes is a literal string. The computer prints strings exactly as is; so it prints YOUR COMPUTER .

The next mark is the semicolon. The semicolon tells the computer to hold its printing right there, so that the next thing it prints follows right on. If the semicolon wasn't there, the computer would move on to the next line before it started printing again. You might like to try the same program, but without the semicolon, just to prove it.

On to line 20. This is pretty simple; it just tells the computer to go to line 10. It must be very boring sometimes, being a computer.

By the way, you can stop and start a running program like this one quite easily. On my computer the control-S key will stop the review program, and the same key will restart it.

So far, we've learnt that the PRINT statement makes the computer print things, that text between quotes is a literal string, and that the semicolon in a PRINT statement makes the 'print head' hold still. And we've been introduced to the GOTO statement.

Here's a revision of what you've learnt so far.

Exercise 1.

1. Make the computer print your name all over the screen.
2. What will this program print?
   10 PRINT "HELLO WORLD! ";
   20 GOTO 20

3 What will this program do?
   10 PRINT HELLO WORLD! ;
   20 GOTO 30

The answers are at the end of the article. Let's press on with something a bit more useful.

Doing Arithmetic

You can make your computer work out sums for you and give you the answers. Let's see how this works. This time, instead of writing programs, we'll use command mode, at least at first. Remember, that means we don't type line numbers, so the computer does what we ask straight away.

Try this: PRINT 2+2

Obviously the answer is 4

Now try this one: PRINT 6.5 − 3.2. Your computer should display 3.3.

Incidentally, if you're getting fed up with typing 'PRINT' again and again, some computers will accept the abbreviation ? instead. Others, notably North Star Basic accept '!'. So: ?6.5 — 3.2 should give you the same answer.

Notice that there's no divide key on the keyboard? And the computer doesn't use × for multiply either. Instead, it uses the / (slash) symbol for division and the * (asterisk) symbol for multiplication.

Let's go forth and multiply. Try:

```
PRINT 4 * 5 + 6
```

Notice that the computer starts reading the line at the left and finishes at the right, same as you and me. First it multiplied 4 by 5, then it added the 6, giving 26. Now try PRINT 6 + 4 * 5

What's this? We got the same answer! Shouldn't it be 6 + 4, that gives 10, times 5 makes 50? Not this time! And this is the reason.

Sure, the computer reads from left to right, but sometimes the order in which it does things is dictated by the rules of arithmetic. For example, as it reads from left to right, it will hold off doing additions and subtractions until it's done any multiplications and divisions, unless there are brackets to force the issue. It doesn't matter which way round you write them, as we've just seen, multiplication and division have a higher priority than addition and subtraction, with brackets having the highest priority of all. Let's see how this works:

Calculation	Steps
6*5+4*7	6*5=30
	4*7=28
	28+30=58
6+5*4+7	5*4=20
	6+20=26
	26+7=33

```
(6+5)*(4+7)        6+5=11
                   4+7=11
                   11*11=121
```

See how, in the absence of brackets, multiplication takes precedence over addition. But with the brackets, the additions in the brackets are done first, and then the multiplication. This is called algebraic hierarchy.

I call it flaming awkward at first, but you'll soon get used to it. It's the way you would normally write down any algebraic formula on paper, which makes it easy to write great long formulae without any ambiguities.

The PRINT statement will work with numbers, as well as strings. The computer can add, subtract, multiply and divide, and it obeys a strict set of rules of priority between these operations. A quick test:

1. PRINT (13 – 7)*6
2. PRINT 24 + 3 * 4
3. PRINT 24 + (* (4 – 2))
4. PRINT "15 + 7 * 3"
5. PRINT 18 / (3 + 6) * 4

Enough of this hard work. The answers are at the end of the article. But you shouldn't need them.

More Printing and Calculating

Let's set about incorporating what we've done so far into a program. We'll work out the circumference of a circle and its area, and print them, with appropriate labelling. We'll assume the radius of the circle is 5cm. Remember the formula for the circumference of a circle is

c = 2 * pi * r

and the formula for the area is

a = pi * r * r

so our program looks like this:

```
NEW
10  PRINT " THE CIRCUMFERENCE
    IS"; 2 * 3.14159 * 5
20  PRINT " THE AREA IS"; 3.14159 * 5
    * 5
30  END
```

Both the first two lines print the string first, followed by the answer to the calculation. Line 30 just marks the end; on most computers it's not needed, and you can leave it out if you prefer.

RUN the program and let's see what we get. On my machine, I got:

```
RUN
THE CIRCUMFERENCE IS 31.4159
THE AREA IS 78.5398
Ok
```

Instead of 'Ok', you may have a prompt of some sort, but otherwise, you should have a pretty similar sort of result. Let's try another example.

To convert Centigrade degrees into Fahrenheit, the formula is: F = 1.8 * C + 32

To convert 37 degrees C to Fahrenheit, our program is really very simple:

```
NEW
10  PRINT "37 DEG. C = ";1.8 * 37 +
    32;"DEG.F"
37 DEG. C = 98.6 DEG. F
Ok
```

Notice how you can mix strings in before and after the answer to the calculation.

Variables

Do you get the impression, somehow, that we're still not using the full power of the computer? You're right, and the things we need to really get going are variables.

A variable is a location inside the computer's memory where it can store values away and recall them when needed.

Every variable has a name, and every time you use a new name, the computer sets away a bit of storage space for it.

There are some rules about what you can call variables. Generally, in the 'standard' BASIC, variable names are either a letter, or a letter followed by a single digit. So these would be okay to use:

A B C Z A1 A7 A9 Z6 Q2

while these would not:

12A 45 NUMBER FRED FRODO

There are several good reasons for this; firstly, to BASIC a number is just a number, unless it's at the beginning of a line, in which case it's a line number.

While the BASIC interpreter might not throw out long names like 'NUMBER', it would just disregard the '—UMBER' part, and confuse it with the variable N you've used elsewhere. Likewise 'FRED' and 'FRODO' would mean the same thing to many BASICs.

Of course, there are exceptions that prove the rule; for example, Microsoft BASIC actually recognises two letters at the beginning of a variable name, and just disregards the rest. Other BASICs use the whole name up to 32 or more characters in length. You might like to experiment and investigate what your BASIC will accept and recognise.

Let's rewrite our circle circumference and area calculator to use a variable:

```
NEW
10  R = 5
20  PRINT "THE CIRCUMFERENCE
    IS" 2 * 3.14159 * R
30  PRINT "THE AREA IS"; 3.14159 * R
    * R
40  END
```

Running this program should give the same result as before. But best of all, to redo the calculation for a circle of 10cm radius, you just retype line 10:

```
10  R = 10
RUN
THE CIRCUMFERENCE IS 62.8318
THE AREA IS 314.159
Ok
```

Now try changing line 10 for other values of R. Calculate the circumferences and areas of circles with radii of 2, 7, 4.638 or 20.0001. Each time you change line 10, the computer can recalculate the figures you want.

Another simple example. Let's print a table of squares.

```
NEW
10  N = 1
20  PRINT N;"SQUARED IS";N * N
30  N = N + 1
40  GOTO 20
```

Run the program and see what you get. Remember, BREAK or control-C will stop it.

By now, you should have a good idea of how the program works; the only line that's a little hair-raising is line 30. When we write 'N = N + 1', we don't actually mean that N is equal to N + 1 or that 9 = 10, or that 2 = 3 or whatever. What we are doing is reading the value stored in N, adding one to it, and then storing our answer back into N again. We are assigning N that new value, which is why '=' is often called the assignment operator.

When you use the assignment operator, bear in mind that you can only have one variable name on the left side of it. The computer can only store the answer to a calculation in one variable at a time.

Now we know about the assignment operator, and what a variable is, and how to name variables. Another short test.

Which of the following lines make sense:

1: 10 PRINT N + 20
2: 25 PRINT 20N
3: 20 N = N * 2
4: 20 * N
5: D = B * B – 4 * A * C
6: GOTO 20 + 5 * N
7: A * B = D

Getting Some Input

There's still a lot more we can do to make that circle calculations program a lot easier to use. The next big step is the INPUT statement.

We already know how to output our results; that's the job of the PRINT statement. But INPUT makes everything so much easier. Here's version three of our circumference/area calculator:

```
NEW
10  INPUT R
```

```
20 PRINT "THE CIRCUMFERENCE
   IS"; 2 * 3.14159 * R
30 PRINT "THE AREA IS ";
   3.14159 * R * R
40 END
```

Try it out. The first thing the computer prints is a question mark.

Now you type in the value of R (say, 5) and hit RETURN.

```
? 5
THE CIRCUMFERENCE IS 31.4159
THE AREA IS 78.5398
```

Now, every time you type RUN, the program will ask you for your input. Or is it the computer that's asking for it? Let's do the same improvement on our Centigrade to Fahrenheit converter.

```
NEW
10 INPUT C
20 PRINT C;"DEG. C =; 1.8 * C +
   32;"DEG. F"
30 END
```

Now try running this program a few times. It should follow the same pattern of stopping to ask you for input, waiting until you hit RETURN, and then printing your answer.

Deluxe INPUT

Leave the C to F converter program in memory, but retype line 10:

```
10 INPUT "WHAT TEMPERATURE
(C)";C
```

Now run the program again. This time, it will ask you for what it wants!

Answers to Exercises

Exercise 1

1. 10 PRINT "your name":
   20 GOTO 10
   I guess we're also forced to accept the command line
   PRINT "your name all over the screen"
2. It prints HELLO WORLD! and then hangs up.
3. The computer prints:
   0
   Undefined line number in 20
   Ok
   (or a similar message), because it thinks 'HELLO WORLD!' in line 10 is the name of a variable, and the target of the goto in line 30 doesn't exist.

Exercise 2

1. 36
2. 26
3. 30
4. 15 + 7 * 3
5. 30
6. 8

Exercise 3

1, 3 and 5 make sense. 2 is wrong because it contains an implied multiplication, which won't work on most machines except some Hewlett-Packard desk-top computers and the Sharp pocket computer. 4 is just meaningless, as it doesn't tell the computer to do anything. In 6, most computers cannot goto the result of a computation (except the Sinclair ZX80 and a few others). In 7, there should only be one variable to the left of the equals sign. Note that although some machines can make sense of some of these, I still call them wrong, as they are not transportable code, and thus not good programming practice. □

Notes

Part 1's INPUT statements were easy — wait till you see the deluxe version. This 'chapter' puts a bit of life into your programming, with the code for a guessing game.

PART II

your computer tutorial

Deluxe INPUT (continued)

WHENEVER the INPUT statement includes a string of letters (in quotes) before a semicolon, it will output the string as a prompt to the user. This makes BASIC programs a lot friendlier, and a whole lot more useful.

For example, our circle calculator program could be changed to ask for its input:

```
10 INPUT "WHAT'S THE RADIUS"; R
20 PRINT "THE CIRCUMFERENCE IS";2 * 
   3.14159 * R
30 PRINT "THE AREA IS"; 3.14159*R*R
40 END
RUN
WHAT'S THE RADIUS? 5
THE CIRCUMFERENCE IS 31.4159
THE AREA IS 78.5398
Ok
```

Notice that BASIC automatically supplies a question mark after the prompt.

Stringing Along

So far, we've been doing a lot of number-crunching. Although computers were originally designed for just that, they're good at other things, too — like handling text.

Just as variables are memory locations, pigeon holes if you like which can hold numbers, so we can have variables that hold text.

These are called string variables, and the text they hold is called a string.

String variables are named in the same way as number variables, except they have a $ sign on the end. So A$, Z$, D9$ and FRED$ are all valid names for string variables. What's more, they are completely different from the numeric variables, A, Z, D9 and FRED.

How are the strings used? As we've already seen, the computer will print literal strings — that is, strings of letters between double quotes ("). So try this:

```
NEW
10 A$ = "HELLO THERE"
20 PRINT A$
RUN
```

Did you get this? You should have:

```
HELLO THERE
Ok
```

The INPUT statement we've already learnt about will work with strings, too. For example:

```
NEW
10 INPUT "HELLO, WHAT'S YOUR NAME";N$
20 PRINT "HI, ";N$;", NICE TO MEET
   YOU"
30 INPUT "HOW OLD ARE YOU";A
40 PRINT "JUST THINK, AT THE TURN OF
   THE CENTURY,"
50 PRINT "YOU'LL BE";A+19;"YEARS OLD"
60 PRINT "SO LONG, ";N$;", NICE
   CHATTING WITH YOU"
70 END
```

Try running this program. You can see how, by just adding more INPUT and PRINT statements, you could make a program that lets you have a conversation with the computer.

Decisions, Decisions...

So far, the operation of our programs has been pretty predictable. It's been a steady progression from one line to the next. Let's start putting a little bit of life, some unpredictability, into our programs, so different things happen at different times. We'll do that with the IF statement. And our first real program should be a game!

```
NEW
10 REM      ***GUESSING GAME VERS 1.1
20 REM       PROGRAMMED IN MBASIC 4.4
30 REM            21/9/81
40 N = INT(99 * RND(1) + 1)
50 PRINT "I'M THINKING OF A NUMBER
   BETWEEN 1 AND 100"
60 PRINT "YOU'VE GOT TO TRY TO GUESS
   IT"
70 INPUT "WHAT'S YOUR GUESS";G
80 IF G > N THEN PRINT "TOO HIGH"
90 IF G < N THEN PRINT "TOO LOW"
```

```
100 IF G = N THEN PRINT "YOU'VE GOT
    IT!!!!!": GOTO 40
110 GOTO 70
120 END
```

There are quite a few new features of BASIC in this program. The first three lines are REMark statements. The computer will ignore them; they're simply there for the benefits of humans reading the program. They give the title and version number, the language used and the date.

Line 40 contains two new functions. The INT(whatever) function takes the integer part of whatever. In other words, it chops off the part after the decimal point. So INT(1.37) equals 1, INT(27.294) equals 27, and INT(0.005) equals 0.

The RND(1) part generates a random number. In other words, it lets the computer 'choose' a number entirely at random. In fact, it's not really random — it only seems that way — and so, to be pedantic, it's a *pseudo-random number.*

The RND function on your computer may be different. On mine, RND(1) generates a random number between 0 and 1, while RND(0) repeats the last random number. On the TRS-80, things are different: RND(0) returns a value between 0 and 1, while RND(X) returns an integer between 1 and X inclusive. It's worth spending a little time experimenting on your computer to find out what its preference is. If it's like the TRS-80, then line 40 can become

```
40 N = RND(100)
```

which is a bit easier to understand. Try working out the operation of the old version, though.

Lines 50 and 60 print the instructions for the game, and line 70 asks the player for his guess.

In line 80, we come across the IF statement. It simply says that if the player's guess is higher than the number, then print a suitable message. In line 90, if the player's guess is too low, another message is printed. Note that only one (or none) of these messages can appear, as the guess can't both be too high and too low!

In line 100, we check to see whether the player was right. This is an example of multiple statements appearing on one line. After the PRINT statement there's a colon, signifying that another statement will follow. If the IF at the beginning of the line is false, then execution will continue with *the next line, not necessarily the next statement.* Consequently, the GOTO 40 will only be executed if the IF statement is true, in other words, if the player's guess is correct.

Let's dig into the theory of the IF statement a bit deeper. Basically, its syntax is IF something (is true) THEN do whatever. For example:

```
IF X = 0 THEN GOTO 720

IF N$ = "FRED" THEN PRINT "SHORT
FOR FREDERICK, I PRESUME"

IF A$ <> "Y" THEN END
```

If the 'do whatever' part of the statement is a GOTO, then the word GOTO can be omitted:

```
IF X = 0 THEN 720
```

If the 'something' part is false, then the program will go on to the next line, ignoring whatever follows the THEN part of the statement.

The content of the 'something' part is what is called a relational operator. These are:

```
>    is greater than
<    is less than
=    is equal to
<>   is not equal to
>=   is greater than or equal to
<=   is less than or equal to
```

For example, the relationship 5 is less than 2 is obviously false. So is the relation 5 is not less than or equal to 2. But 5 is greater than 2 is true, as is 5 is greater than and not equal to 2. Relational expressions can be quite complex. Try this one for size:

```
X * 5 / 2 + Y >= 6 * Y - 2 * 3 / Z
```

Is it true or false, when X = 3, Y = 4 and Z = 6?

It's false. The left hand side calculates out at 11.5, while the right side is 23.

Loopin' de Loop

We've already seen how, using the GOTO instruction, we can make a program go round and round in circles. But there's an even easier way to do it. Suppose we want to calculate a factorial.

A factorial is a number multiplied by every integer smaller than itself, down to zero. For example, factorial 5 (written 5!) is:

5! = 1 × 2 × 3 × 4 × 5 (= 120)

Similarly, 10! is:

10! = 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 x 10 (= 3628800)

Let's write a program to calculate this, using the GOTO statement.

```
10 REM      ***FACTORIAL PROGRAM***
20 REM      WRITTEN IN MBASIC 4.4
30 REM      21/9/81
40 INPUT "NUMBER";N
50 X = 1: F = 1
60 IF X=N THEN PRINT
   "FACTORIAL";N;"=";F:END
70 X = X + 1
80 F = F * X
90 GOTO 60
```

This program works, but it's not terribly elegant. It took two attempts to get it to run correctly (I forgot to set F = 1 first time around) and I still don't like it.

BASIC provides a statement which is ideal for controlling loops. It's called the FOR . . . NEXT statement, and it's really a pair of statements — one at the top of the loop and one at the end. Let's look at an example (and clean up the factorial program at the same time!):

```
10 REM  **FACTORIAL PROGRAM USING FOR**
20 REM      WRITTEN IN MBASIC 4.4
30 REM      21/9/81
40 INPUT "NUMBER";N
50 F = 1
60 FOR X = 1 TO N
70 F = F * X
80 NEXT X
90 PRINT "FACTORIAL";N;"=";F
100 END
```

This version ran first time. It's a lot easier to see the structure of the program too.

There's a loop, clearly outlined by the FOR statement a the top, and the NEXT statement at the bottom.

This is what happens: The first time the computer encounters the FOR statement, it sets the loop counter (X in this example) to its initial value (1 here), and then proceeds into the loop body. When it reaches the NEXT statement, it tests the loop counter, to see if it has reached its terminating value (N in this example). And if it hasn't, it returns to the top of the loop and increments the loop counter.

If the loop counter has reached its terminating value, the computer carries on from the bottom of the loop.

The basic form of FOR . . .NEXT loop is

```
FOR (initial condition) TO
(terminating condition)
                .
            (loop body)
                .
NEXT (loop counter)
```

It is possible to specify an increment other than 1 (?) by using the format

FOR (blah, blah) TO (blech, blech) STEP (amount)

For example, try this:

```
10 REM      *** SINE CURVE PLOTTER ***
20 REM      WRITTEN IN MBASIC 4.4
30 REM      21/9/81
40 W = 64
50 H = 16
60 FOR A = 0 TO 6.28 STEP 6.28/H
70 PRINT TAB(W/2 + W/2 * SIN(A));"*"
80 NEXT A
90 GOTO 90
```

This is an example of a simple curve plotting program. Line 40 should be set to the width of your terminal, and line 50 to the depth.

Line 60 is the first part of the FOR . . .NEXT loop. It loops from 0 to 6.28, in steps of 6.28H, so there will be H steps.

If you have a 16-line terminal, the loop increment will be 0.3925. Line 70 does the plotting: the TAB(n) function moves the cursor n spaces across the screen, so the first W/2 moves it half-way across the screen (to centre the plot).

The remainder of the formula takes the sine curve, which varies from −1 to 1, and multiplies it by half the screen width, to provide the complete TAB position.

The PRINT statement then moves the cursor, and prints the '*' at the correct place. This is a useful trick to remember if you want to produce plots from calculated data, but don't have 'full' graphics capabilities. Of course, the graph comes out sideways, but it's better than nothing!

Line 90 of the program is simply a 'dynamic halt' which holds the computer up to prevent it printing 'OK' and lousing up the pretty pattern on the screen!

Convenience Functions

We're now at the stage of writing some fairly lengthy programs, and unless you're an accomplished typist, you may be starting to feel the strain! Microsoft BASIC has a number of functions which are designed to make life easier for you.

For example, it can be a bit of a strain remembering to enter line numbers, but MBASIC has a function to do that for you. Try typing this:

```
AUTO
    REM *** PROGRAM TO GENERATE
    6 RANDOM NUMBERS ***
    REM WRITTEN IN MBASIC 4.4
    REM 23/9/81
FOR N = 1 TO 6
PRINT "RANDOM NUMBER"; N ;" IS: "; RND(1)
NEXT N
```

Then hit the Control C key to stop the automatic line-numbering. If you don't have a control key, then you should hit 'BREAK'. If you don't have a break key, you'd better check in the manual!

The computer should respond with its usual prompt, eg 'Ok'.

Now type LIST. You should see:

```
10      REM *** PROGRAM TO GENERATE
        6 RANDOM NUMBERS ***
20      REM WRITTEN IN MBASIC 4.4
30      REM 23/9/81
40 FOR N=1 TO 6
50 PRINT "RANDOM NUMBER"; N ;" IS: ";
   RND(1)
60 NEXT N
```

The AUTO command will generate line numbers as required. For example, the command "AUTO 100,20" will generate line numbers starting at 100 and increasing by 20's. If AUTO comes to a line number which is already allocated to a line, it will print an asterisk after the line number. It carries on only if you want to overwrite previously existing lines, otherwise hit control C or BREAK.

Now, suppose we want to add extra lines to our program. How do we do this without starting from scratch?

Take the example of printing a random bar chart, rather than random numbers.

First, we delete line 50 (simply by typing in 50 — it replaces the old line with nothing), then we add 6 new lines:

```
35 W = 64   'TERMINAL WIDTH
45 V = INT(W * RND(1)) + 1
47 FOR I = 1 TO V
50 PRINT "*";
52 NEXT I
55 PRINT
```

The resulting program should look like this:

```
10      REM *** PROGRAM TO GENERATE
6 RANDOM NUMBERS ***
20      REM WRITTEN IN MBASIC 4.4
30      REM 23/9/81
35 W = 80
40 FOR N=1 TO 6
45 V = INT(W * RND(1)) + 1
47 FOR I = 1 TO V
50 PRINT "*";
52 NEXT I
55 PRINT
60 NEXT N
```

Try running this modified version of the program. There's still something wrong.

Being Pedantic

Pedantic as it might seem, the documentation of a program is very important, and we have the title of our program wrong. It should say 'Program to draw 6 random bars', so change it.

There are two ways of doing this. As we've already seen, we could simply retype it. But this is tedious, and we could make mistakes. Instead, we can edit it in order to correct it.

Here's how. Type EDIT 10 and the computer will respond with the line number and nothing else (for the moment).

Now we hit the space bar, and the computer will type a character at a time across the screen. Keep hitting the space bar until the computer has typed:

```
10 REM      *** PROGRAM TO
```

And include the space after the word 'TO'. Now we want to delete the word. 'GENERATE', which is eight characters long. Type '8D', and you should see:

```
10 REM      *** PROGRAM TO \GENERATE\
```

The next step is to insert the word 'DRAW, by typing:

```
IDRAW
```

Nothing happens when you type I. But following characters will be put into the line. To terminate the insert made, use the ESC (escape) key. □

In this instalment Les looks at saving programs, subroutines, how to handle arrays, and the difference between a command, a statement and a function . . . while Brendan Akhurst explores the possible side effects of fancying yourself a hacker.

PART III

Deletions

NOW WE can step over to the word NUMBERS and type:

```
7DIBARS<ESC>
```

```
where <ESC> means the ESC key.
```

There are other subcommands in the edit mode. For example, the X command moves the cursor to the right end of the line and enters insert mode, so that you can add onto the line. The H command will 'hack off' the rest of the line to the right of the cursor, and enter insert mode so that you can retype sections of a line.

The S command will search for a character. If preceded by a digit n, it will search for the nth occurrence of the character. So 3SP will move the cursor to the third P in the line, and print all the characters before it. The K command kills all characters before the specified character, so: 3KP would delete every character up until the third P, and wait for the next command.

Several commands are provided to exit edit mode. Normally, pressing CR will print the remainder of the line and exit edit mode,E just will exit, and Q will scrap any alterations and leave edit mode with the original line intact. A does the same thing as Q, but automatically restarts edit mode to give you another bite at the cherry, while L prints the balance of the line, and restarts edit mode.

The C command allows the user to change the next n characters to something else. The command 4CGOTO would delete the next four characters, replacing them with the word GOTO. Beware! The number of characters you insert *MUST* be the same as the number you delete, otherwise strange things will happen!

your computer tutorial

Finally, to delete characters behind the cursor, just use the DEL key.

Saving Programs

Now that we can edit programs, we will be tackling much larger projects, and it would be nice if we could save our programs on cassette or disk to save us retyping them.

Cassette interfaces, and the way they work, vary enormously from machine to machine. In general, the procedure is this:

Plug in the cassette recorder, according to the computer handbook, and switch it on. Insert a blank cassette, and move the tape forward past the leader and onto the magnetic material. If your cassette recorder doesn't move, you may have to temporarily unplug the "pause" plug (usually the smallest).

With all cables connected, and the blank tape at the right place, type CSAVE or, if your computer allows it, CSAVE "filename", and hit RETURN. The tape should start to move for a few seconds as the computer saves the program on tape. Once it has finished, rewind the tape ready for reloading.

To load the program into the computer is a similar procedure. Move the tape to a point just before the start of the program, insert all cables, and type CLOAD or optionally CLOAD "filename", and hit RETURN.

The computer should now load the program off the tape. It may be necessary to experiment with the volume control when loading, as most cassette interfaces are sensitive to the input level.

If you type CLOAD "filename", most computers will read in the next program if it has the right name, otherwise it will skip over programs until it finds the right one. If you put multiple programs on a tape, jot down readings from the tape index counter to mark where programs start,so that you can reposition the tape to the right place later.

Saving programs on disk systems is much easier and quicker.

Just type SAVE "filename", and the computer will save the file to disk. It can be reloaded by typing LOAD "filename".

You may be able to load and run a program in one operation by typing RUN "filename".

Functions

As well as the statements and commands we've already learnt (and there are more to come), BASIC provides things called functions. In this section, we're going to learn what these functions do. But first, what's the difference between a command, a statement, and a function?

A command is an instruction to the computer to do something immediately. For example, RUN or LIST or AUTO.

A statement is a line of code, or part of a line, which the computer will store for later execution. When executed, it provides the instructions which the machine will follow.


...HA... I'M GETTING RIGHT INTO IT NOW...
...I'M FINALLY ON TOP OF THIS MACHINE... ...HEH HEH HEH....
HEH HEH
YES BOSS
JAB!
...I KNOW ALL ABOUT RND'S AND functions
BELL'S BIBLE OF BABBLE
...WORDS AREN'T SIMPLY WHAT THEY APPEAR TO BE... MISSPELT...
...THEY HAVE HIDDEN MEANINGS ONLY A COMPUTER EXPERT LIKE MYSELF CAN FATHOM... HEH HEH HEH...
=SNORT=...COMPUTING IS A NEW... NO... THE NEW WAY OF LIFE... HEH.. HACKING!... HACKER... THAT'S WHAT I AM...
....TO BE ABLE TO CAUSE A MILLION SMALL ELECTRICAL IMPULSES TO function ACCORDING TO MY... HEH... Command... POWER!
K commands!
NO MASSA PLEAZ DOAN....
...EXCUSE ME, BUT YOUR WIFE..
WIFE?
...WIFE?... WIFE... wife... ...WIFE?...
wife? what them thing?
TAP TAP TAP TAP TAP
WIFE... wife... wi... ...W... W... V... b... a... w...
LES' LITERARY LIABILITY
COULD YOU CALL BACK? ...HE'S JUST HAVING AN nth occurrence OF THE THIRD KIND...
FZZT POP FUSE BURN
YES, I'LL PULL HIS PAUSE PLUG...
©BJ. AUGUST '81

It's rather like a sentence, in that it contains a verb (GOTO, INPUT, PRINT) and some objects which the verb acts upon, either variable names (X, A, Z9), constants (1, 3.14159) or line numbers (250, 80010).

A function acts upon some other value or variable to provide a new value. It is said to **return** that value. For example, you might want the sine of an angle. BASIC provides a function to perform this calculation, thus:

210 L = SIN(A)

Notice that we set L equal to the value returned by the function.This is a good way to distinguish functions from statements. For example, you can't say:

250 L = GOTO 330

because GOTO 330 doesn't have a value, that is it's not a function.

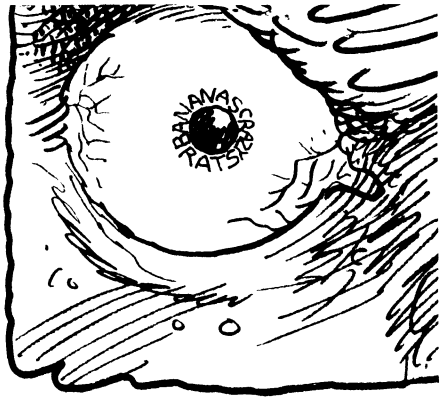
Here are some of the functions commonly provided in BASIC:

ABS(X)	absolute magnitude of X; if X < 0 then ABS(X) = -X
ASN(X)	arcsine of X (rarely seen)
ACS(X)	arccosine of X (rarely seen)
ATN(X)	arctangent of X
COS(X)	cosine of X
EXP(X)	transcendental number e raised to the power of X
INP(X)	input from port X
INT(X)	integer part of X
LOG(X)	common logarithm of X
PEEK(X)	read the contents of memory location X
RND(X)	random number
SIN(X)	sine of X
SQR(X)	square root of X
SQRT(X)	same as SQR(X) (rarely seen)
TAB(X)	produces a string of X spaces

Don't worry about what these functions do — that's mathematics, not computer programming. If your intention is to use the computer to solve mathematical problems, then I guess you know what they do already, otherwise you may only need the INP, INT, PEEK, RND and TAB functions, which aren't strictly mathematical. INP and PEEK will be explained later, and RND you've met already.

INT returns the integer part of a number, so for example, INT(27.34) is 27, INT(3.141592654) is 3. Watch out for negative numbers, though – INT(3.141592654) is – 4. This is because the definition of INT is the "largest integer less than the original number." Therefore, – 3 is GREATER than – 3.141592654, not less.

TAB simply produces a string of spaces. So TAB(5) prints 5 spaces, and TAB(15) prints 15 spaces. This is particularly useful if you're laying out printed reports. And it's particularly useful if you're



trying to copy a program from a printer page. Which is easier to follow:

```
10 PRINT "                    HAMURABI"
```

or

```
10 PRINT TAB(17);"HAMURABI"
```

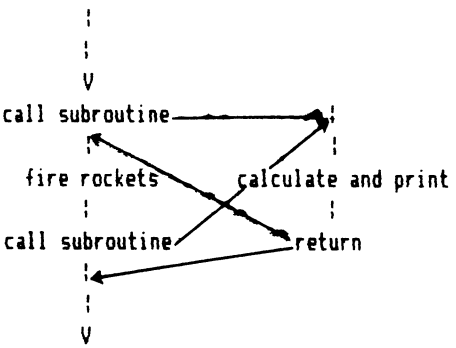
They both do the same thing, but the TAB function spells it out clearly.

Programs within Programs

Quite often, we find within a program that we have blocks of instructions that perform the same job at two or more points in a program. Now, computer programmers have a phrase to describe that kind of situation — they call it "reinventing the wheel". After all, if there was a way to use the same set of instructions from different points within a program, that would make life a lot easier, with fewer typing mistakes and shorter programs.

Well, there is a way to do this, and in programming parlance it's called a **subroutine.** For example, suppose we have a program which simulates the flight of a spacecraft around a planet, and that we want the program to calculate and print the height of the spacecraft both before and after it fires its maneouvering rockets.

Here's one way to do it:



Calling a Subroutine

In BASIC, there is a special keyword that is used to call subroutines. The statement: GOSUB lineno will transfer control to the given line number, until such time as a RETURN statement is executed, when normal execution resumes with the next statement after the GOSUB. So, in practice, if this was a program, it would be laid out like this:

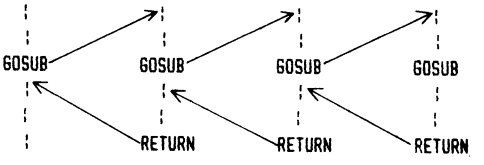
```
10 REM     SUBROUTINE DEMO
20 REM
.
.
150 GOSUB 1000
160 LET . . .
.
.
250 GOSUB 1000
260 GOTO . .
.
.
990 END
1000 REM ** CALCULATE AND PRINT POSITION **
1010 LET . . .
.
.
1080 RETURN
```

Notice how the program is organised. The flow is quite naturally from top to bottom, and, as long as we know what the subroutine at line 1000 does, it makes it easy to follow the main program. The main program ENDs before the subroutine, in order to avoid accidentally executing it. This is not mandatory, but it is good programming style.

Now let's look at what happens when the computer comes to a GOSUB. It knows that although it is about to jump to another part of the program, it's got to come back to this part when it's done, so it remembers the line number of the next statement that it would normally execute. That's the one it will continue with when it RETURNs. It sticks this onto a list of such line numbers called a **stack,** and then goes wherever the GOSUB commands.

When the subroutine has done its job, the computer will come to a RETURN statement. This tells it to take the return address off the stack again, and go back to it. (By the way, the computer uses the stack to hold line numbers during FOR-NEXT loops too — can you see why?).

The stack is an interesting kind of memo pad, in that the computer can stick as many line numbers on it as it likes, but it can only read them back out in reverse order to the way they were written. This means that whenever a RETURN is executed, it terminates the most recently called subroutine, and the next RETURN finishes the one before that, and so on. We call subroutines within subroutines **nested subroutines.**



Arrays

Before we start writing programs that use subroutines, let's get into another useful BASIC structure, — arrays.

Quite often, we find it useful to present a collection of related figures in the form of a table. For example, we might collect rainfall figures over a year:

```
Month: ! Jan ! Feb ! Mar ! Apr ! May ! Jun ! Jul ! Aug ! Sep
-------------------------------------------------------------
Rain   ! 4" ! 3" ! 3" ! 5" ! 5" ! 5" ! 8" ! 7" ! 6"
```

BASIC allows us to store these related values in one, admittedly complex, variable, called on array. Arrays are named like ordinary variables, except that they have an associated number, called a subscript. So, if A is an array, the A(5) is the sixth entry in the array (don't forget A(0)!). For our rainfall chart, we could have

```
A(0) ! A(1) ! A(2) ! A(3) ! A(4) ! A(5) ! A(6) ! A(7) ! A(8)
-------------------------------------------------------------
  4  !  3  !  3  !  5  !  5  !  5  !  8  !  7  !  6
```

The first element in the array is A(0), which has a value of 4, the second is A(1) which as a value of 3, and so on.

Some BASICs start arrays with the zeroth element, whereas some start with the first; this is something to check up on. A few offer the user the option, by using a statement like

OPTION BASE 0 (or OPTION BASE1)

to let the user set it up the way he prefers. There's an old joke that says if you ask someone to count up to 10 they'll go: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 — except someone who works with computers. He'll go: zero, 1, 2, 3, etc.

Array variables can be used in assignment statements in exactly the same way as ordinary variables, so you can say:

360 A(6) = X + B(6)

Before you use an array in a program, if it will have more than 10 elements in it, you must inform the BASIC interpreter, so it can reserve space for the array. You do this with the DIMension statement:

40 DIM A(32), B(32), M(64)

which will reserve storage space for three arrays, two of 32 elements and one of 64.

Arrays are particularly potent when used with loops. Suppose you've got an array of 12 values which must be input. This can be done by putting the input statement inside a loop:

```
10 DIM V(12)
20 FOR I = 1 TO 12
30 PRINT "VALUE";I;
40 INPUT V(I)
50 NEXT I
```

Th' 'op will ask for a value 12 times an' .e the values in succeeding ele- r .s of the array V. You can print arrays ✓ a similar fashion.

String Arrays

It is possible to have arrays of strings (in Microsoft BASIC — sorry, NorthStar owners). We'll show an interesting use of string arrays later. Let's look first at how arrays can be used and at the same time, we'll start to look at the use of subroutines — our programs are becoming big enough to make use of them.

The problem we're going to solve is quite a common one around computers — it's the kind of job they're good at. Let's suppose we want to sort a sequence of numbers into increasing numerical order. There are several ways to do it; we'll use the Standard Exchange Sort, otherwise known as the Bubble Sort.

The idea behind Bubble Sort is very simple. We start at the bottom of an array and compare the first two numbers. If the first is larger than the second, then we swap them. Otherwise, we just leave them as they are. We now move on to the second and third numbers and repeat the process, then the third and fourth,and so on.

After we've completed this sequence for every pair of adjacent numbers in the array, we will have moved the highest number into the last location of the array, and we can now repeat the procedure, which will move the second highest into the second last location. So, to sort n numbers, we have to make n-1 passes through the array. We can save some time, however, by noticing that once the last location is right, we don't have to compare it, and the same applies to the second-last, third-last and so on as we know they are in the right place.

If this seems complicated, follow this example:

Array at start:
5 2 3 6 1
First compare, and swap:
2 ⟷ 5 3 6 1
Second compare and swap:
2 3 ⟷ 5 6 1
Third compare, no swap:
2 3 5 ⟷ 6 1
Fourth compare and swap: (puts highest number in last location)
2 3 5 1 ⟷ 6
Start of second pass, first compare, no swap
2 ⟷ 3 5 1 6
Second compare, no swap:
2 3 ⟷ 5 1 6
Third compare, swap:
2 3 1 ⟷ 5 6

No need for a fourth compare, as we know last element is right.

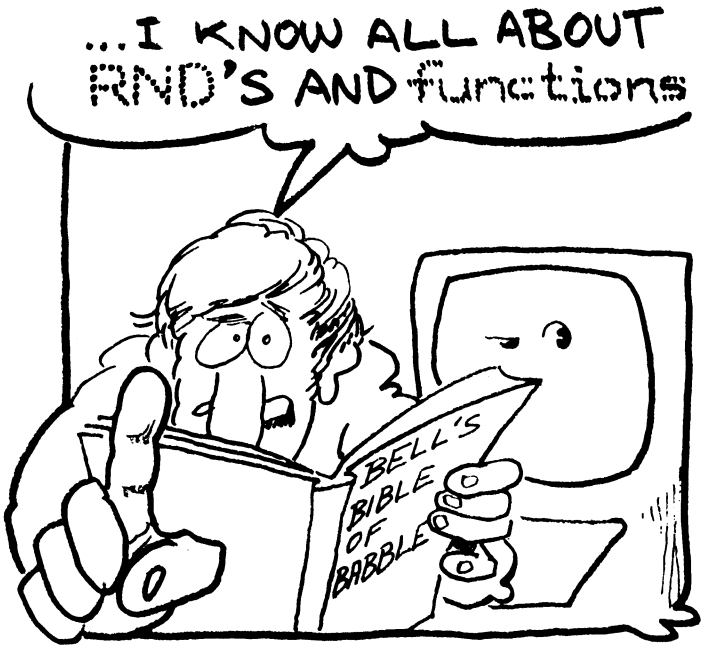
Start of third pass, first compare, no swap:
2 ⟷ 3 1 5 6
Second compare, swap:
2 1 ⟷ 3 5 6

No need for a third compare, as we know fourth element is right.

Fourth pass, first compare and swap:
1 ⟷ 2 3 5 6

No need for a second compare, as we know third element is right. This completes the sorting of the array.

Now here's a program which does the same thing, based on a simple exchange sort algorithm:



```
100 REM ### SORT DEMONSTRATION V1.0 ###
110 REM     WRITTEN IN MBASIC 4.4
120 REM     27/10/81
130 DIM V(100)
140 PRINT "SORT DEMONSTRATION"
150 REM INPUT CONTENTS OF ARRAY,
    ENDING WITH ZERO
160 N=1
170 INPUT V(N)
180 IF V(N) = 0 THEN 210
190 N=N+1
200 GOTO 170
210 N = N - 1
220 :
230 REM START SORTING
240 X = 0
250 FOR I = 1 TO N - 1
260   IF V(I) > V(I+1) THEN GOSUB 370
270 NEXT I
280 IF X = 0 THEN 310
290 GOTO 240
300 REM PRINT RESULTS
310 FOR I=1 TO N
320   PRINT V(I),
330 NEXT I
340 END
350 :
360 REM SWAP SUBROUTINE
370 T = V(I)
380 V(I) = V(I+1)
390 V(I+1) = T
400 X = 1
410 RETURN
```

Line 130 dimensions an array with 100 elements, and then lines 150 to 200 allow us to input the members of the array. Because we don't know in advance how many numbers we're going to input, we've used an old programming trick of using an unusual value to terminate the input. In other words, the program will always accept a zero as the last element, and jump to the sort routine proper.

This sort keeps making complete passes through the array, setting a flag (X) each time it makes an exchange (line 400), and resetting it to zero at the beginning of a pass (line 240). If the flag is zero at the end of a pass (line 280) then there were no swaps in that pass, and the array must be in order.

Lines 310 to 340 simply print out the array. Notice the comma in line 320, at the end of the print statement. This is a variation on the semicolon we've been so careful about, and its effect is to print the results in **fields,** each 14 characters wide. This is very useful for producing tables of ouput.

Notice also our use of the GOSUB to line 370. This enables us to put our series of instructions which perform the swap tidily out of the way at the end of the program.

Now, remembering what we said earlier about not having to complete a pass because we knew the larger values were already in their correct places at the end of the array, let's rewrite the sort, and tidy it up:

```
100 REM  ### SORT DEMONSTRATION V1.1 ###
110 REM      WRITTEN IN MBASIC 4.4
120 REM      27/10/81
130 DIM V(100)
140 PRINT "SORT DEMONSTRATION"
150 REM INPUT CONTENTS OF ARRAY,
    ENDING WITH ZERO
160 N=1
170 INPUT V(N)
180 IF V(N) = 0 THEN 210
190 N=N+1
200 GOTO 170
210 N = N - 1
220 :
230 REM START SORTING
240 FOR P = 1 TO N-1
250   FOR I = 1 TO N-P
260     IF V(I) > V(I+1) THEN
        SWAP V(I),V(I+1)
270   NEXT I
280 NEXT P
300 REM PRINT RESULTS
310 FOR I=1 TO N
320   PRINT V(I),
330 NEXT I
340 END
```

Several things combine to make this a neater program. Firstly, the algorithm is expressed more neatly, using a counter P (line 240) to control the number of passes through the array, while the I counter still indexes through the array. Secondly, I only runs up as far as N– P, stopping short of the end of the array, because, as we've pointed out, there is no need to go that far. Thirdly, we've replaced the swap subroutine with Microsoft BASIC's SWAP statement — okay, we admit it! We were holding out on you all along!

Here's some work for you to do. Type in both the sort programs into your computer and run them both with the following data:

```
15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
2  1  3  4  5  6  7  8  9  10 11 12 13 14 15 0
1  2  3  4  5  6  7  8  9  10 11 12 13 15 14 0
```

Which is faster, and why? Under what circumstances is each faster? □

According to Les, all the interesting things you can do with computers involve strings. Here he cuts them to the right lengths and wraps up your original sort program. As the cartoon characters conclude, "He's a man to be admired."

PART IV

FROM NOW on, we're going to concentrate much more on strings, because all the interesting things you can do with computers involve strings. Just think of the applications for a computer in the office: word processing, accounting, maintaining production schedules, sorting lists of names and addresses. . .

Now just a moment. Let's see if we can sort strings the same way we sorted numbers. Go back to V1.1 of our sort program and see what has to be changed. Our array, V(100), should obviously be an array of strings, called V$(100). Apart from that, everything should be okay. Or will it?

There are a couple of things that will need changing. First, the name of the program should be changed, and the date. Next, our method of detecting the end of input should be changed. Instead of detecting a zero as the end, let's change it to an empty line (two double quotes with nothing between them). Go ahead and do that; your result should look like this:

```
100 REM      ### NAME SORT V1.0 ###
110 REM      WRITTEN IN MBASIC 4.4
120 REM      23/11/81
130 DIM V$(100)
140 PRINT "NAME SORT DEMO"
150 REM INPUT CONTENTS OF ARRAY,
     ENDING WITH A BLANK LINE
160 N=1
170 INPUT V$(N)
180 IF V$(N) = "" THEN 210
190 N=N+1
200 GOTO 170
210 N = N - 1
220 :
230 REM START SORTING
```


```
240 FOR P = 1 TO N-1
250   FOR I = 1 TO N-P
260     IF V$(I) > V$(I+1) THEN
          SWAP V$(I),V$(I+1)
270   NEXT I
280 NEXT P
300 REM PRINT RESULTS
310 FOR I=1 TO N
320   PRINT V$(I),
330 NEXT I
340 END
```

Try it out. It'll work okay, as long as you don't put in more than about 15 names. Then, all of a sudden, you'll get an 'Out of String Space in 170' or 'OS' error code. What's gone wrong?

Microsoft BASIC allocates a certain amount of memory space to string storage. In this case, we've used it all up. To get more, we must add a CLEAR statement to our program:

```
125 CLEAR 1000
```

This will set aside 1000 bytes (or characters) of storage for our string array — enough for 100 names, at an average 10 characters each.

Notice our comparison in line 260 still works, as does the SWAP instruction (if you don't have the SWAP instruction, use a GOSUB to a modified swapping subroutine). We can expand this still further. Here's the beginnings of a mailing list program for a computer club:

```
100 REM      ### MAIL LIST V1.0 ###
110 REM      WRITTEN IN MBASIC 4.4
120 REM      23/11/81
130 CLEAR 5000
140 DIM N$(100), A1$(100), A2$(100), PC(100)
150 PRINT "MAILING LIST"
160 REM INPUT NAMES AND ADDRESSES,
      ENDING WITH A BLANK LINE
170 N=1
180 INPUT "NAME    :";N$(N)
190 IF N$(N) = "" THEN 260
200 INPUT "ADDR1   :";A1$(N)
210 INPUT "ADDR2   :";A2$(N)
220 INPUT "POSTCODE:";PC(N)
230 PRINT
240 N=N+1
250 GOTO 180
260 N = N - 1
270 :
280 REM START SORTING
290 FOR P = 1 TO N-1
300   FOR I = 1 TO N-P
310    IF N$(I) > N$(I+1) THEN GOSUB 440
320   NEXT I
330 NEXT P
340 REM PRINT RESULTS
```

...YEP... COMPLETELY BURNT OUT...
...I'LL DO EVERYTHING I CAN...
...TELL HIS KIN NOT TO WORRY...

SPIK T ME SAY SUMMINK YR HERTIN MI KEY BORD DOAP...

..HELP ME GET HIM INTO THE BACK OF MY HEARS... PANEL VAN...

GOSH! LES BELL HAS BROAD SHOULDERS...

YEAH... A REAL SCOUT...

INDEED A MAN TO BE ADMIRED...

SINCE LAST ISSUE, THE ABOVE SCENE HAS BEEN REPEATED MANY...NO... MANY, MANY, MANY TIMES IN LOTS...UH... LOTS AND LOTS OF PLACES.*....

ANYWAY...

WHA... WHE... !!? AM I DEAD?...UH.. ...ARE YOU GOD?...

I'M LES BELL...

...I'VE JUST SAVED YOUR LIFE...

...OH... I OWE Y.....

...YOU DON'T OWE ME A SINGLE THING...

MY REWARD IS SEEING YOU EVENTUALLY RECOVER...

*THIS WORD BALLOON COURTESY OF THE RONALD REAGAN SCHOOL OF SIMPLE TALK

WOBBLE SQUEE SQUEEK

...NOW, HERE'S A "THERAPEUTIC" COMPUTER CLUB MAILING LIST FOR YOU TO SORT...

HUMMM...

*WARNING: 10 MILL VOLTS ELECTRIC

SUNNY BELL ESTATE
HOME FOR BURNT OUT HACKERS (PTY. LTD.)

ARTARMON DIVISION OF BELL INTERNATIONAL COMPUTER CLUBS A/ASIA INC. ETC.

IS THIS STORY TRUE..? HAVE ONLY THE FACTS BEEN CHANGED TO PROTECT US... MR BELL? ..IS THERE AN END TO THIS PARANOIA?... SHOULD THERE BE A SENATE SELECT ROYAL JUDICIAL PARLIAMENTARY INDEPENDENT BRIBE?.....

```
350 FOR I=1 TO N
360   PRINT N$(I)
370   PRINT A1$(I)
380   PRINT A2$(I);PC(I)
390 PRINT
400 NEXT I
410 END
420 :
430 REM SWAP SUBROUTINE
440 T$ = N$(I)
450 N$(I) = N$(I+1)
460 N$(I+1) = T$
470 T$ = A1$(I)
480 A1$(I) = A1$(I+1)
490 A1$(I+1) = T$
500 T$ = A2$(I)
510 A2$(I) = A2$(I+1)
520 A2$(I+1) = T$
530 T = PC(I)
540 PC(I) = PC(I+1)
550 PC(I+1) = T
560 RETURN
```

In this version of the program, I haven't used the SWAP statement, as it's not available on TRS-80s and some other small computers. This program prompts the user for names and addresses, and once the operator has entered all the data he just hits RETURN when the NAME: prompt appears. The program then sorts the names and addresses and outputs them in alphabetic order.

Note that names should be entered surname first and initials last, *without any commas*, as BASIC will think you are trying to enter two variables with a comma between them.

Improving the Product

There are several things we can do to make our sort program better. Some are dependent on the features of BASIC and some are just straight computer science tricks that will work on any computer in any language.

The first thing to notice is we're spending a lot of time and energy swapping things around in that subroutine at the end. First, we swap the names over, then the first lines of the addresses, then the second line of the address and then, finally, the postcode. All of this takes time — swapping strings is particularly slow. And just think how much slower it will become if we start adding extra information, like what kind of computer our club member owns, when his membership expires, and so on.

There's a better method. We create an array which contains *pointers* to the entries in our data arrays, and we call this array an index.

Let's see how this works. Initially, our index contains just the integers from 1 to 100 in ascending order. When we compare two names in the data arrays and find they're out of order, instead of swapping them, we swap *the corresponding entries in the index array*. This is a lot quicker. Each entry in the index tells us the *position* of the corresponding entry in the data array.

An Index, of A Sort

Of course, our data array is still in the same order as when we started — only the index has changed. So we can't just print it out. Instead, we must look each name up in the index in order to find its position in the data array. Here's how it looks:

BEFORE		AFTER	
index array	data	index array	data
1	9	4	9
2	7	3	7
3	4	2	4
4	3	1	3

Before sorting, each entry in the index is in order, and the nth entry in the index corresponds to the nth entry in the data array. After sorting, the first entry in the index corresponds to the lowest valued entry in the data array, which is probably not the first.

This enables us to do away with that clumsy swapping subroutine from line 370 onwards, with a huge saving in time.

Second, many of the numbers we're dealing with are whole numbers; that is, they don't have any significant digits after the decimal point. Such numbers are called *integers*, and BASIC can treat them as a special case, with a consequent increase in speed and decrease in storage requirements.

To refer to a value as an integer, just add a % sign to the variable name. By going through our program and renaming loop counters like I and N to I% and N%, we can further speed up the program.

We should also note the entries in our index array are integers, so it should be an integer array, with a big space saving.

Finally, the sort algorithm (set of rules) we used is none too efficient itself. The bubble sort moves the high numbers to the right end quite quickly. But if low numbers are far out of place, it takes a long time to move them down. For this reason, the bubble sort is slow.

Shellsort Bursts the Bubble

In the next version of the program, I've replaced the bubble sort with a much faster sort: the Shellsort (D A Shell, *A high-speed sorting procedure*, Communications of the Association for Computing Machinery No 2, 1959; pp30-32).

The Shellsort is much faster. There are three parts to any sorting program (excluding the input and output subsections of course) — the comparison, the swapping method, and the algorithm itself.

In this case, we've changed the swapping method first of all (I tested an indexed version of the program with the old bubble sort first), then the algorithm. At each stage, the program still worked and could be tested. This is called stepwise refinement, and it is a keystone of *structured programming*.

Here's the souped-up version of the program:

```
100 REM   ### SUPER MAIL LIST V1.0 ###
110 REM   WRITTEN IN MBASIC 4.4
120 REM   26/11/81
130 CLEAR 5000
140 DIM IA%(100), N$(100), A1$(100),
      A2$(100), PC%(100)
150 PRINT TAB(22);"MAILING LIST":PRINT
160 REM INPUT NAMES AND ADDRESSES,
      ENDING WITH A BLANK LINE
170 N%=1
180 IA%(N%)=N%
190 INPUT "NAME    :";N$(N%)
200 IF N$(N%) = "" THEN 270
210 INPUT "ADDR1   :";A1$(N%)
220 INPUT "ADDR2   :";A2$(N%)
230 INPUT "POSTCODE:";PC%(N%)
240 PRINT
250 N%=N%+1
260 GOTO 180
270 N% = N% - 1
280 :
290 REM START SORTING
300 FOR I%=1 TO N% STEP I%
310   M% = 2 * I% - 1
320 NEXT I%
330 K% = N% - M%
340 FOR J% = 1 TO K%
350   FOR I% = J% TO 1 STEP -M%
360       IF N$(IA%(I%)) > N$(IA%(I%+1))
          THEN SWAP IA%(I%),IA%(I%+1)
370   NEXT I%
380 NEXT J%
390 M% = M% / 2
400 IF M% <> 0 THEN 330
410 REM PRINT RESULTS
420 FOR I%=1 TO N%
430   PRINT N$(IA%(I%))
440   PRINT A1$(IA%(I%))
450   PRINT A2$(IA%(I%));PC%(IA%(I%))
460 PRINT
470 NEXT I%
480 END
```

Watch the birdie! This chapter finds Les looking at ways to file away (and retrieve) data. The resulting program is a useful mailing list utility, so if nothing else you can treat it as a Pocket Program and lift it for your own use.

PART V

WE ALREADY know how to save and load programs. But our programs are going to be dealing with larger and larger quantities of data, and it's a good idea to have some way of storing that data on tape or disk so we can dispense with all this typing. This brings us to the concept of **data files**.

Data files are the same as the filing cabinets in every office, except they're more efficiently organised (I define a filing system as 'a system for losing things in alphabetical order').

The computer likes to know ahead of time what kind of information is going to be where. It doesn't know the difference between a name and an address, so if you tell it 'every fourth string will be a name, the two strings after it are address lines, and the final piece of data is a post code', you'd better keep your promise to the computer and organise the file that way.

Here's a few terms of data file parlance. A complete collection of related information, such as a name, address and postcode, is called a **record**. Within the record each item of information such as name, address line 1, address line 2 and postcode, is called a **field**.

On some computers, and in certain circumstances, each record must be the same length and the fields are of predefined length too. That may seem like a lot of bother, but it does confer certain advantages as we'll see. In the meantime, we're going to start with the lazy man's type of data files, called **sequential files**.

In sequential files, the computer doesn't care how long your fields and records are; it just reads them one after another, and the beginning of every field comes right after the end of the preceding one.

Here's how it works: imagine you wanted to make an entry in a conventional paper file (actually, computer data files are 'conventional'; paper is passe). How would you do it?

You'd first of all work out where the file should be in the filing cabinet and locate it. Then you'd pull it out and open it. Then you read through the file quickly until you found the end, and then you'd make your entry. Finally, you would close your file and replace it ready for the next occasion.

Yes, The Same Way

You use exactly the same technique to access a sequential data file. First, you locate the file. Okay, so there aren't any alphabetical tabs on a floppy disk. But it's the operating system's job to keep track of what files are where, so you can rely on it to do that part of the job. Next, you open it with the command OPEN "I",1,"filename".

The OPEN statement requests the operating system to locate this file and keep tabs on it while you rummage through the contents. It also signals BASIC to set up a 'file pointer', which initially points to the beginning of the file, but will move through it as you read or write the file.

The 'I' part indicates that you are opening this file for input; you're going to be reading from it. An 'O' would mean you were going to output to the file. Sequential files can be open for input or output, but not both.

This raises a minor problem. Opening a file for output sets the file pointer to the beginning of the file, so anything you write onto the file will just overwrite the previous contents. And you can't switch from reading to writing, thus eliminating the possibility of reading through until the end and then appending the new information (although CBASIC-2 allows this). So how can you append on to a file?

The answer is to read from the original file, and output a copy of this to a new file. After the first file has been read, you can forget about it and write your appended data onto the new copy of the file. So far so good. Let's get back to that OPEN statement.

Different BASICs allow varying numbers of files to be open at one time, typically up to 15. At sign-on, TRS-80 Disk BASIC for example asks you how many files you will be using. If you don't answer, it assumes a maximum of three.

Each file is allocated a number, when we open it. In this case, we've said our file will be file number 1. Finally, the last part of the OPEN statement is the filename itself, which can be either a string constant or a string variable.

Thinking time. Write OPEN statements to do the following:

1. Open file DEALER.DAT as file number 3 for input.
2. Open file RAINFALL.FIG for output, as file number 2.


...WE ALREADY KNOW HOW TO SAVE & LOAD PROGRAMS...
©B.J.AKHURST.82
...THIS BRINGS US TO THE CONCEPT OF DATA FILES...
HOME
...HEH...REMEMBER LAST MONTH WE TALKED ABOUT, AH, COMPUTER CLUB MAILING LISTS...
THESE PREMISES PATROLLED BY SAVAGE GUARD DOGS
MAILING ROOMS
SNAP!
WELL, WE MUST NOW BE ABLE TO STORE, RETRIEVE...& ADD TO THESE LISTS...
ALL CORRECT SIR...
GARD
HELP! SEND HELP TO ARTARMON! PLEASE!
UNCHAIN US!
SOLIDARITY!
I WANNA GO HOME...
MAIL ROOM
EXCUSE ME
FLABBA DABBA DABBA DABBA
TROMPA TROMPA TROMPA TROMPA
... ANYHOW, READ THE TEXT... AH... WORK IT ALL OUT... AND.. ..UH...
HELP!
CRUSH
....NEXT MONTH, WE'LL MOVE ONTO SOMETHING NEW!...

3. Open file SALEFIGS.DAT for input, as file number 1.
4. Our program has already asked for the filename and stored it in variable F$. Open the file for input, as file 1.

Dear Files, I'm Writing...

Writing to files is as easy as writing to the screen. Having assigned a file number to each open file, we now use a modification of the PRINT statement to write into it.

PRINT#1,A,B,C for example, will print the values of A, B and C into the file (in the usual 14-character wide fields). The statement PRINT #1,A","B","C will print the three numbers, separated by commas, which is the best format to use if the numbers are to be read by a subsequent (you guessed it) INPUT#1 statement.

Obviously TRS-80 owners cannot use the PRINT@ statement to print to a file, because a file is only one-dimensional. It's a strip of characters, in sequence (hence sequential file). It consists purely of the data we write to it, with added spaces inside fields in cases where we separate printed values with commas.

It is possible to use PRINT #n USING to print to a data file though, as it simply formats the output for neatness. There are occasions where this could be useful.

Once the last data item has been written to a file, it is **closed**. This means the operating system writes out the last data to the disk, and updates the file's directory entry to reflect its new length. This is done using the CLOSE n statement, where n is the number of the file to be closed. Here's a simple example of this process:

```
270 OPEN "O",1,          'Open file for output
   "DATAFILE.DAT"
280 FOR N=1 TO 20        'Loop 20 times
290 PRINT#1 A(N);",";    'Output the variable
300 NEXT N
310 CLOSE 1              'Close the file
```

This segment of code writes the 20 values of A(1) through A(20) out to the disk, separated by commas.

The CLOSE statement can close more than one file at a time, simply by specifying the file numbers separated by commas. If no numbers are given, the CLOSE statement will close all open files.

Reading A Sequential File

Now we have the 20 variables stored in a disk file, we will want to read them back. How is this done? It's exactly the same as writing, except the file is opened for input, and we use the INPUT#n statement. So:

```
130 OPEN "I",1,
   "DATAFILE.DAT"   'Open file for input
140 FOR N=1 TO 20   'Set up loop
150 INPUT#1,A(N)    'Input variable
160 NEXT N
170 CLOSE 1         'Close file
```

Thinking time again. Write code to write 20 names and addresses onto a disk file called NAD.DAT.

```
430 OPEN "O",1,"NAD.DAT"
440 FOR N=1 TO 20
450 PRINT#1,N$(N);",";A1$(N);",";
    A2$(N);",";PC%(N)
460 NEXT N
470 CLOSE 1
```

Now write code to read them off again.

```
720 OPEN "I",1,"NAD.DAT"
730 FOR N=1 TO 20
740 INPUT#1,N$(N);",";A1$(N);",";
    A2$(N);",";PC%(N)
750 NEXT N
760 CLOSE 1
```

An answer is given at the end of the article, but the ultimate test is, of course, whether your computer will do it.

Now, To End The File

The examples tackled above work fine as long as you know in advance how many records you will be reading. But what happens if you don't know how long a file is, but just keep looping around, reading it? The answer is that as soon as you've read the last item of data in a file, the next time you try to read from it you will get an error message, and your program will stop. Not good.

BASIC gets around this problem by providing a flag called EOF, which stands for End Of File. This is automatically set to true when you read the last data item in a file. Now we can include a test for the EOF flag in our read loop, and everything will be fine:

```
370 OPEN "I",1,"NAD.DAT"
380 N=1
390 IF EOF(1) THEN 440
400 INPUT#1, N$(N),A1$(N),A2$(N),PC%(N)
410 N=N+1
420 GOTO 390
430 REM CONTINUE PROCESSING
440 CLOSE 1
```

If we were inputting from file 2, then we would test for EOF(2). For file 5 it would be EOF(5), and so on.

The Mailing List Program

With all this in mind, it is now time for us to complete our mailing list program. In particular, we will need to add several functions to our basic sort utility. We have to be able to:

1. Add names to the file.
2. Delete names from the file.
3. Sort the file into either alphabetic order or postcode order.
4. List the file to the screen.
5. Print labels from the file.

Additional functions might be used to specify the name of the file we are working on and to exit back to the operating system. In particular, I have applied one restriction to this program to make it more useful in the 'commercial' environment. It is to be compatible with MicroPro's MailMerge utility, part of the WordStar word processing package.

We'll start with the overall system design, using a technique known as **flowcharting.** The chart shows the overall operation of the program and relates the various routines. The routines each perform one of the basic functions referred to above. Each is a separate functional block in the main program.

The program starts, as usual, with its name and historical information, followed by the declarations which reserve string space and dimension the arrays.

```
100 REM     ### SEQUENTIAL FILE
              MAILING LIST MANAGER ###
110 REM     ### COMPATIBLE WITH
              MICROPRO MAILMERGE ###
120 REM     WRITTEN IN MBASIC 4.4
130 REM     1/12/81
140 :
150 CLEAR 10000
160 DIM IA%(100), N$(100), C$(100),
    A1$(100), A2$(100), PC%(100)
```

This is all pretty straightforward; there's nothing new for us here. Next, we start the program by printing its name on the screen and asking for the name of the file to work on.

```
170 PRINT CHR$(12);TAB(22);"MAILING LIST"
    : PRINT: PRINT
180 PRINT: INPUT "FILE TO WORK ON";F$
190 :
```

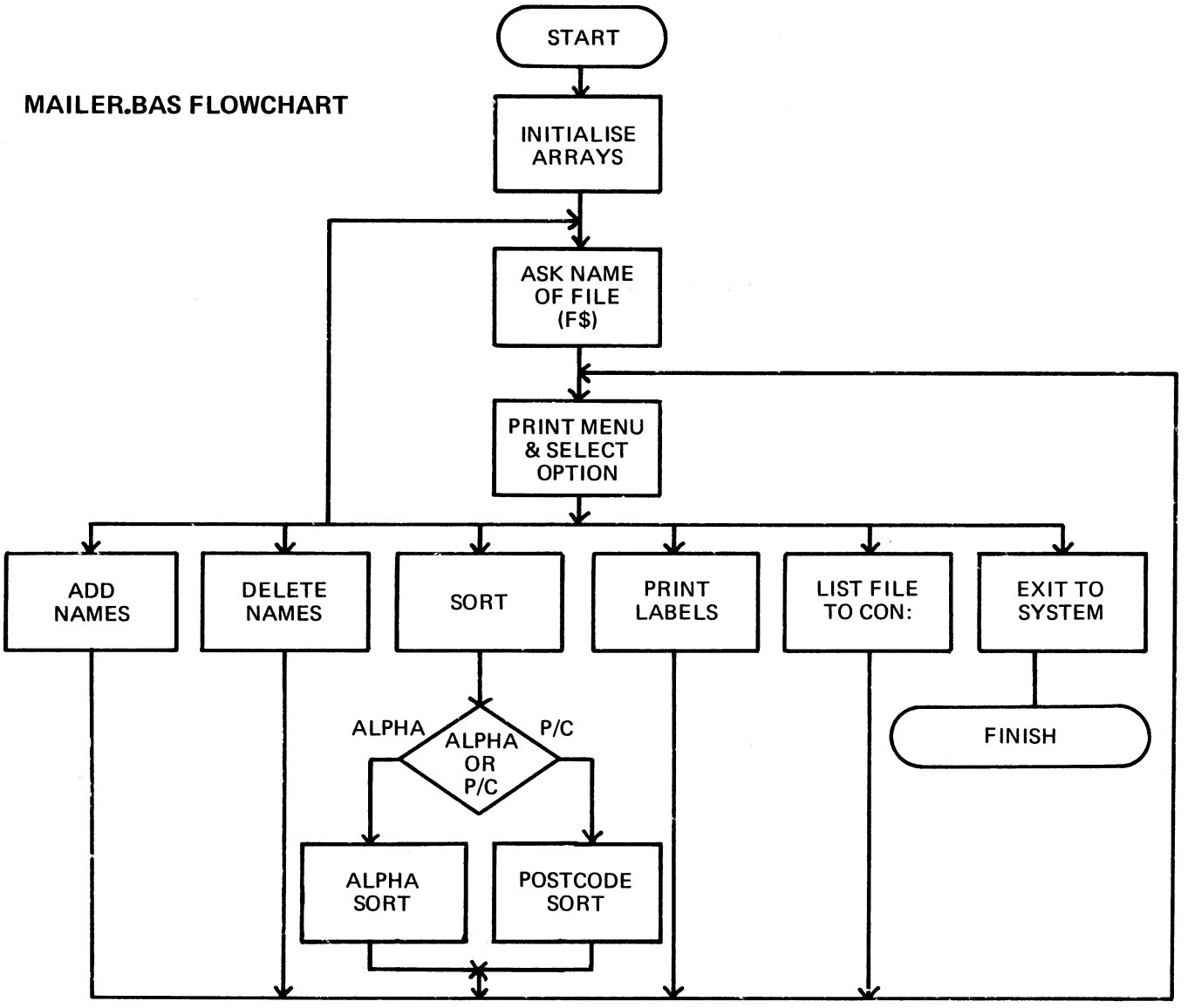
None of this is very startling, either. CHR$(12) is the character that clears the screen on my terminal. TRS-80 owners will want to replace the CHR$(12) with CLS.

Could We See The Menu?

Having cleared the screen and input the name of the file we're going to work on, we are now ready to offer the user a choice of things to do. We'll do this by presenting a **menu** of options. Here is the code:

```
200 REM          ### DISPLAY MENU ###
210 :
220 PRINT CHR$(12);"1 - Add Names"
230 PRINT "2 - Delete Names"
240 PRINT "3 - Sort File"
250 PRINT "4 - List File to CON:"
260 PRINT "5 - Print Labels"
270 PRINT "6 - Change Work File"
280 PRINT "7 - Quit and Return to CP/M"
290 PRINT: INPUT "Enter Choice";C
300 IF C<1 OR C>7 THEN 220
```

MAILER.BAS FLOWCHART




```
310 ON C GOTO 380,1060,490,1740,
    1480,170,340
320 :
```

Again, the screen is cleared, then a list of seven options is displayed. Line 300 re-presents the list if the user types in an answer outside the expected range. Line 310 is our first use of the ON...GOTO statement. This uses the value in the specified variable (in this case C) to select one arm of a multi-way branch. If C is 1, control is passed to the first line number of the list, if it is 2 we jump to the second, and so on.

The easiest subroutine is going to be number 7. Here it is:

```
330 REM EXIT MBASIC
340 SYSTEM
350 :
```

TRS-80 owners would probably replace the SYSTEM statement with END.

The next subroutine to be tackled is the one that adds names and addresses to the file.

Remember our earlier discussion of how it is impossible to read and write from the same sequential file. In this case, I have chosen to keep the original file but rename it from 'filename.DAT' to 'filename.BAK'. This way, the user can recover from any serious errors by simply erasing the new version of the file and renaming the .BAK version to the correct name.

Here is the routine:

```
360 REM **ROUTINE TO ADD NAMES TO FILE**
370 :
380 PRINT CHR$(12);"Add Names to File ";
    F$: FOR N=1 TO 300: NEXT N
390 I%=1:GOSUB 1250:I%=I%-1:GOSUB 1360
400 IF EOF(1) THEN 440
410 INPUT #1,N$,C$,A1$,A2$,PC
420 PRINT #2,N$;",";C$;","A1$;",";
    A2$;",";PC
430 GOTO 400
440 FOR N%=1 TO I%
450 PRINT #2,N$(N%);",";C$(N%);",";
    A1$(N%);",";A2$(N%);",";PC%(N%)
460 NEXT N%
470 CLOSE 1,2:GOTO 220
480 :
```

You will notice this routine uses two subroutines. Here is the first of these:

```
1230 REM *** SUBROUTINE TO BUILD ARRAY
          OF NAMES AND ADDRESSES ***
1240 :
1250 N$(I%)="":PRINT CHR$(12):PRINT:
     PRINT: INPUT "Name      :";N$(I%)
1260 IF N$(I%) = "" THEN RETURN
1270 INPUT "First name:";C$(I%)
1280 INPUT "Addr1     :";A1$(I%)
```

```
1290 INPUT "Addr2     :";A2$(I%)
1300 INPUT "Postcode  :";PC%(I%)
1310 I% = I% + 1
1320 GOTO 1250
1330 :
```

I% is set by the calling program to be 1. This subroutine simply loops around, inputting a surname, given name, address lines and postcode. Whenever no name is input, it returns to the calling program.

Here is the subroutine which renames the original file to .BAK and opens the files:

```
1340 REM ### SUBROUTINE TO RENAME AND
          CREATE BACKUP FILE ###
1350 :
1360 R$=LEFT$(F$,LEN(F$)-3)+"BAK"
1370 ON ERROR GOTO 1430
1380 KILL R$
1390 NAME F$ AS R$
1400 OPEN "I",1,R$
1410 OPEN "O",2,F$
1420 RETURN
1430 IF ERR = 53 THEN RESUME NEXT
1440 PRINT "Strange Error in ";ERL:STOP
1450 :
```

Line 1360 uses the LEFT$ function to 'grab' all but the right-most three characters of F$ (the filename) and then sticks .BAK on the end. In line 1380 there is a possibility of an error occurring: it doesn't really bother us if there isn't an existing '.BAK' file, as we are going to KILL (or erase) it anyway.

Nonetheless, MBASIC will report a 'File Does Not Exist' error and drop us out of our program. We get around this by line 1370, which directs MBASIC to jump to line 1430 if it finds an error.

Once there, if the error is coded number 53 ('File Does Not Exist'), then we just ignore it and resume execution with the next statement (that is, line 1390). Otherwise, we print an error message, including the line where the error was found.

Normal execution opens the two files in exactly the way we've learned.

Arrays Are The Key

Now the operation of the main 'Add Names' routine will become a little clearer. It works by building up an array of names and addresses in memory. This is done by the first subroutine. Once that is complete, it renames the input and output files and opens them. It then copies the input file across to the output file and outputs the contents of the array on to the end. Finally, both files are closed. It's that simple.

Next, we'll tackle the sort routines. First, we print a little menu to let the user decide which sort of sort he/she wants:

```
490 REM          ### SORT SUB-SYSTEM ###
500 :
510 PRINT CHR$(12);"Sort Sub-system"
520 PRINT:PRINT:PRINT "1 - Sort on Name"
530 PRINT "2 - Sort on Postcode"
540 PRINT:INPUT "Enter Choice";C
```

Having decided that, we then read in the file:

```
550 GOSUB 1360
560 N%=1
570 IF EOF(1) THEN CLOSE 1: GOTO 620
580 IA%(N%)=N%
590 INPUT #1, N$(N%),C$(N%),A1$(N%),
    A2$(N%),PC%(N%)
600 N%=N%+1
610 GOTO 570
620 N% = N% - 1: PRINT: PRINT: PRINT
    N%;"Records Read": PRINT
```

Now the file is in memory, we jump to the appropriate sort routine:

```
630 ON C GOTO 670,910   'DECIDE WHICH
                         SORT TO USE
640 :
```

Here is the sort on surname:

```
650 REM      ### SHELL SORT ON NAME ###
660 :
670 FOR I%=1 TO N% STEP I%
680   M% = 2 * I% - 1
690 NEXT I%
700 K% = N% - M%
710 FOR J% = 1 TO K%
720   FOR I% = J% TO 1 STEP -M%
730     IF N$(IA%(I%)) > N$(IA%(I%+1))
        THEN SWAP IA%(I%),IA%(I%+1)
740   NEXT I%
750 NEXT J%
760 M% = M% / 2
770 IF M% <> 0 THEN 700
780 :
```

As you can see, this is exactly the same sort routine we used before. And having completed the sort, we write the file back to the output file:

```
790 REM      ### WRITE OUTPUT FILE ###
800 :
810 FOR I%=1 TO N%
820   PRINT #2, N$(IA%(I%));",";
830   PRINT #2, C$(IA%(I%));",";
840   PRINT #2, A1$(IA%(I%));",";
850   PRINT #2, A2$(IA%(I%));",";
      PC%(IA%(I%))
860 NEXT I%
870 PRINT I%-1;"Records Written":
    CLOSE 2:GOTO 220
880 :
```

This is very straightforward. For those who appreciate small subtle differences, here's the postcode sort:

```
890 REM   ### SHELL SORT ON POSTCODE ###
900 :
910 FOR I%=1 TO N% STEP I%
920   M% = 2 * I% - 1
930 NEXT I%
940 K% = N% - M%
950 FOR J% = 1 TO K%
960   FOR I% = J% TO 1 STEP -M%
970     IF PC%(IA%(I%)) > PC%(IA%(I%+1))
        THEN SWAP IA%(I%),IA%(I%+1)
980   NEXT I%
990 NEXT J%
1000 M% = M% / 2
1010 IF M% <> 0 THEN 940
1020 GOTO 810
1030 :
```

As you see, the two sorts are remarkably similar.

The next routine to be tackled is the one which deletes names from the file. The approach I have chosen here is to minimise file accessing by deleting up to a hundred names in one go. Here's the routine:

```
1040 REM ### ROUTINE TO ERASE NAMES ###
1050 :
1060 PRINT CHR$(12):PRINT TAB(24);
     "Delete Names":FOR N = 1 TO 300:NEXT N
1070 PRINT CHR$(12)
1080 I%=1:GOSUB 1250:I%=I%-1:GOSUB 1360
1090 IF EOF(1) THEN 1160
1100 INPUT#1,N$,C$,A1$,A2$,PC%
1110 FOR N%= 1 TO I%
1120   IF N$(N%) = N$ AND PC%(N%) = PC%
       THEN 1180
1130 NEXT N%
1140 PRINT #2,N$;",";C$;",";A1$;",";
     A2$;",";PC%
1150 GOTO 1090
1160 CLOSE 1,2
1170 GOTO 220
1180 PRINT CHR$(12);"Match found:":PRINT
1190 PRINT N$:PRINT C$:PRINT A1$:
     PRINT A2$;",";PC%:PRINT
1200 INPUT "Delete (Y/N)";A$:IF LEFT$(A$,1)
     = "Y" OR LEFT$(A$,1)="y" THEN 1090
1210 GOTO 1130
1220 :
```

The routine starts off the same way as the 'Add Names' routine, by building an array of names and addresses and opening the input and output files. Then the two diverge.

Finding A Match

As each record is input from the input file (line 1100), it is compared with each

BASIC FOR BIRDWATCHERS

name and postcode in the array (lines 1110-1130). If they match the program jumps to line 1180; otherwise the record is just written out to the output file. Finally, the files are closed and we are returned to the main menu.

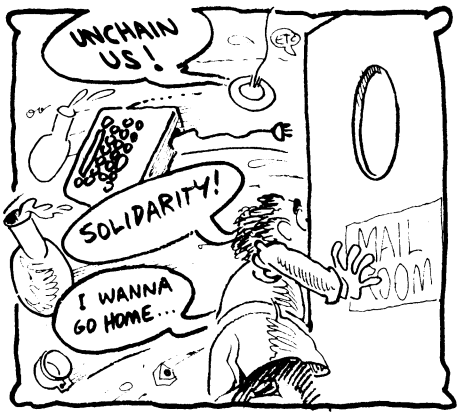
If a match is found, the record is printed (line 1190) and the user asked if the record is to be deleted. If it is, execution continues with the reading of the next record. If not, then it is output in the normal way.

The most important function of a mailing list program is to print labels which are then used to address envelopes. Here's the routine to do that:

```
1460 REM ** SUBROUTINE TO PRINT LABELS **
1470 :
1480 PRINT CHR$(12):PRINT:PRINT:
     PRINT TAB(30);"Now printing labels."
1490 OPEN "I",1,F$
1500 IF EOF(1) THEN 1520
1510 N%=1:INPUT#1,N$(1),C$(1),A1$(1),
     A2$(1),PC%(1)
1520 IF EOF(1) THEN 1540
1530 N%=2:INPUT#1,N$(2),C$(2),A1$(2),
     A2$(2),PC%(2)
1540 IF EOF(1) THEN 1560
1550 N%=3:INPUT#1,N$(3),C$(3),A1$(3),
     A2$(3),PC%(3)
1560 LPRINT TAB(4);:FOR I%=1 TO N%
1570   LPRINT C$(I%);
1580   IF C$(I%)<>"" THEN PRINT" ";
1590   LPRINT N$(I%);TAB(I%*26+4);
1600 NEXT I%:LPRINT
1610 LPRINT TAB(4);:FOR I%=1 TO N%
1620   LPRINT A1$(I%);TAB(I%*26+4);
1630 NEXT I%:LPRINT
1640 LPRINT TAB(4);:FOR I%=1 TO N%
1650   LPRINT A2$(I%);",";PC%(I%);
       TAB(I%*26+4);
1660 NEXT I%:LPRINT:LPRINT:LPRINT
1670 IF EOF(1) THEN 1690
1680 GOTO 1500
1690 CLOSE 1:LPRINT CHR$(12)
1700 GOTO 220
1710 :
```

This job isn't as easy as it looks on first thought. The trick is to read in three records at a time, then print the three names across the page, followed by the three first address lines and so on. However, if the end of file is reached with only one or two records to be printed across the page, this complicates matters.

In this routine, I used N% to count the



number of records that are read in each 'pass' and this then controls the FOR..NEXT loops that print across the page.

Apart from that there's nothing complicated about this routine. It just needs slow, careful reading to make its operation evident. In line 1690, the CHR$(12) is a form feed character, causing the printer to page eject.

Finally, a similar routine is needed to list the file to the console. In this routine I took a slightly different approach:

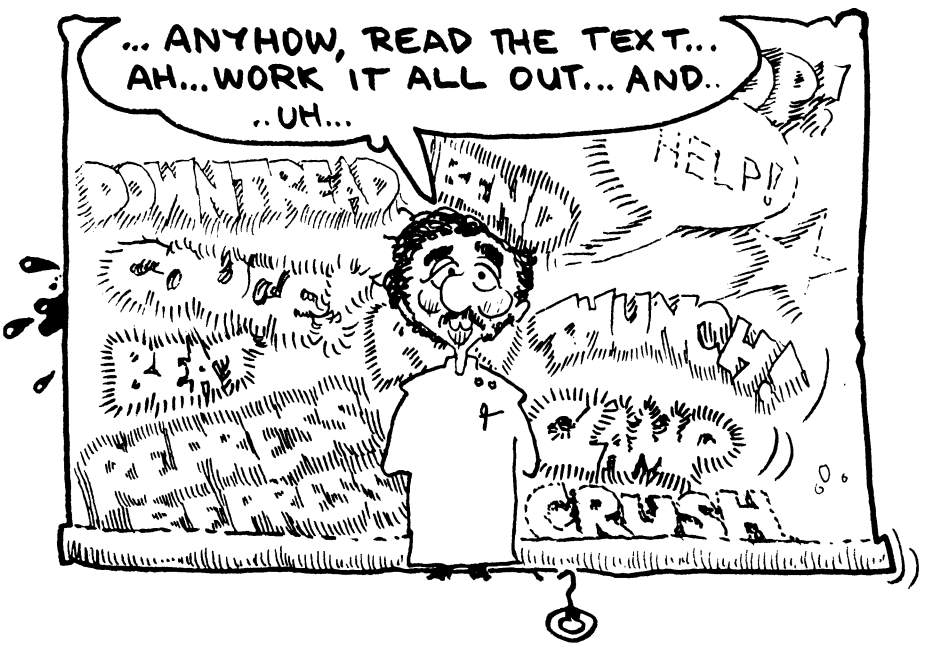
```
1720 REM ** SUBROUTINE TO LIST TO CON: **
1730 :
1740 PRINT CHR$(12):PRINT:PRINT:
     PRINT TAB(30);"File Listing:"
1750 WIDTH 80
1760 OPEN "I",1,F$
1770 IF EOF(1) THEN 1790
1780 N%=1:INPUT#1,N$(1),C$(1),A1$(1),
     A2$(1),PC%(1)
1790 IF EOF(1) THEN N$(2)="":C$(2)="":
   A1$(2)="":A2$(2)="":PC%(2)=0:GOTO 1810
1800 N%=2:INPUT#1,N$(2),C$(2),A1$(2),
     A2$(2),PC%(2)
1810 IF EOF(1) THEN N$(3)="":C$(3)="":
   A1$(3)="":A2$(3)="":PC%(3)=0:GOTO 1830
1820 N%=3:INPUT#1,N$(3),C$(3),A1$(3),
     A2$(3),PC%(3)
1830 PRINT C$(1);
1840 IF C$(1)<>"" THEN PRINT " ";
1850 PRINT N$(1);TAB(26);C$(2);
1860 IF C$(2)<>"" THEN PRINT " ";
1870 PRINT N$(2);TAB(52);C$(3);
1880 IF C$(3)<>"" THEN PRINT " ";
1890 PRINT N$(3)
1900 PRINT A1$(1);TAB(26);A1$(2);
     TAB(52);A1$(3)
1910 PRINT A2$(1);",";PC%(1);TAB(26)
     ;A2$(2);
1920 PRINT ",";PC%(2);TAB(52);
     A2$(3);",";PC%(3)
1930 PRINT:PRINT
1940 IF EOF(1) THEN 1960
1950 GOTO 1770
1960 CLOSE 1:WIDTH 72:INPUT "Hit return
     to continue";A$
1970 GOTO 220
1980 END
```

This routine is considerably simpler than the label printer, and you should have no trouble following it.

Finally, to make it easier to key into your own computer, here is the complete program:

```
100 REM      *** SEQUENTIAL FILE
               MAILING LIST MANAGER ***
110 REM      *** COMPATIBLE WITH
               MICROPRO MAILMERGE ***
120 REM      WRITTEN IN MBASIC 4.4
130 REM      1/12/81
140 :
150 CLEAR 10000
160 DIM IA%(100), N$(100), C$(100),
    A1$(100), A2$(100), PC%(100)
170 PRINT CHR$(12);TAB(22);"MAILING
    LIST":PRINT
180 PRINT: PRINT: INPUT "FILE TO WORK
    ON";F$
190 :
200 REM          *** DISPLAY MENU ***
210 :
220 PRINT CHR$(12);"1 - Add Names"
230 PRINT "2 - Delete Names"
240 PRINT "3 - Sort File"
250 PRINT "4 - List File to CON:"
260 PRINT "5 - Print Labels"
270 PRINT "6 - Change Work File"
280 PRINT "7 - Quit and Return to CP/M"
290 PRINT: INPUT "Enter Choice";C
300 IF C<1 OR C>7 THEN 220
310 ON C GOTO 380,1060,490,1740,1480,
    170,340
320 :
330 REM EXIT MBASIC
340 SYSTEM
350 :
360 REM ** ROUTINE TO ADD NAMES TO FILE **
370 :
380 PRINT CHR$(12);"Add Names to
    File ";F$: FOR N=1 TO 300: NEXT N
390 I%=1:GOSUB 1250:I%=I%-1:GOSUB 1360
400 IF EOF(1) THEN 440
410 INPUT #1,N$,C$,A1$,A2$,PC
420 PRINT #2,N$;",";C$;","A1$;",";
    A2$;",";PC
430 GOTO 400
440 FOR N%=1 TO I%
450 PRINT #2,N$(N%);",";C$(N%);",";
    A1$(N%);",";A2$(N%);",";PC%(N%)
460 NEXT N%
```

```
470 CLOSE 1,2:GOTO 220
480 :
490 REM         ### SORT SUB-SYSTEM ###
500 :
510 PRINT CHR$(12);"Sort Sub-system"
520 PRINT:PRINT:PRINT "1 - Sort on Name"
530 PRINT "2 - Sort on Postcode"
540 PRINT:INPUT "Enter Choice";C
550 GOSUB 1360
560 N%=1
570 IF EOF(1) THEN CLOSE 1: GOTO 620
580 IA%(N%)=N%
590 INPUT #1, N$(N%),C$(N%),A1$(N%),
    A2$(N%),PC%(N%)
600 N%=N%+1
610 GOTO 570
620 N% = N% - 1: PRINT: PRINT: PRINT
    N%;"Records Read": PRINT
630 ON C GOTO 670,910
640 :
650 REM   ### SHELL SORT ON NAME ###
660 :
670 FOR I%=1 TO N% STEP I%
680   M% = 2 * I% - 1
690 NEXT I%
700 K% = N% - M%
710 FOR J% = 1 TO K%
720   FOR I% = J% TO 1 STEP -M%
730     IF N$(IA%(I%)) > N$(IA%(I%+1))
        THEN SWAP IA%(I%),IA%(I%+1)
740   NEXT I%
750 NEXT J%
760 M% = M% / 2
770 IF M% <> 0 THEN 700
780 :
790 REM     ### WRITE OUTPUT FILE ###
800 :
810 FOR I%=1 TO N%
820   PRINT #2, N$(IA%(I%));",";
830   PRINT #2, C$(IA%(I%));",";
840   PRINT #2, A1$(IA%(I%));",";
850   PRINT #2, A2$(IA%(I%));",";
      PC%(IA%(I%))
860 NEXT I%
870 PRINT I%-1;"Records Written":
    CLOSE 2:GOTO 220
880 :
890 REM ### SHELL SORT ON POSTCODE ###
900 :
910 FOR I%=1 TO N% STEP I%
920   M% = 2 * I% - 1
930 NEXT I%
940 K% = N% - M%
950 FOR J% = 1 TO K%
960   FOR I% = J% TO 1 STEP -M%
970     IF PC%(IA%(I%)) > PC%(IA%(I%+1))
        THEN SWAP IA%(I%),IA%(I%+1)
980   NEXT I%
990 NEXT J%
1000 M% = M% / 2
1010 IF M% <> 0 THEN 940
1020 GOTO 810
1030 :
1040 REM ### ROUTINE TO ERASE NAMES ###
1050 :
1060 PRINT CHR$(12):PRINT TAB(24);
     "Delete Names":FOR N = 1 TO 300:NEXT N
1070 PRINT CHR$(12)
1080 I%=1:GOSUB 1250:I%=I%-1:GOSUB 1360
1090 IF EOF(1) THEN 1160
1100 INPUT#1,N$,C$,A1$,A2$,PC%
1110 FOR N%= 1 TO I%
1120   IF N$(N%) = N$ AND PC%(N%) = PC%
       THEN 1180
1130 NEXT N%
1140 PRINT #2,N$;",";C$;",";A1$;",";
     A2$;",";PC%
1150 GOTO 1090
1160 CLOSE 1,2
1170 GOTO 220
1180 PRINT CHR$(12);"Match found:":PRINT
1190 PRINT N$:PRINT C$:PRINT A1$:PRINT
     A2$;",";PC%:PRINT
1200 INPUT "Delete (Y/N)";A$:IF LEFT$(A$,1)
     ="Y" OR LEFT$(A$,1)="y" THEN 1090
1210 GOTO 1130
1220 :
1230 REM ### SUBROUTINE TO BUILD ARRAY
          OF NAMES AND ADDRESSES ###
1240 :
1250 N$(I%)="":PRINT CHR$(12):PRINT:
     PRINT: INPUT "Name      :";N$(I%)
1260 IF N$(I%) = "" THEN RETURN
1270 INPUT "First name:";C$(I%)
1280 INPUT "Addr1     :";A1$(I%)
1290 INPUT "Addr2     :";A2$(I%)
1300 INPUT "Postcode  :";PC%(I%)
1310 I% = I% + 1
1320 GOTO 1250
1330 :
1340 REM      ### SUBROUTINE TO RENAME
          AND CREATE BACKUP FILE ###
1350 :
1360 R$=LEFT$(F$,LEN(F$)-3)+"BAK"
1370 ON ERROR GOTO 1430
1380 KILL R$
1390 NAME F$ AS R$
1400 OPEN "I",1,R$
1410 OPEN "O",2,F$
1420 RETURN
1430 IF ERR = 53 THEN RESUME NEXT
1440 PRINT "Strange Error in ";ERL:STOP
1450 :
1460 REM ## SUBROUTINE TO PRINT LABELS ##
1470 :
1480 PRINT CHR$(12):PRINT:PRINT:
     PRINT TAB(30);"Now printing labels."
1490 OPEN "I",1,F$
1500 IF EOF(1) THEN 1520
1510 N%=1:INPUT#1,N$(1),C$(1),A1$(1),
     A2$(1),PC%(1)
1520 IF EOF(1) THEN 1540
1530 N%=2:INPUT#1,N$(2),C$(2),A1$(2),
     A2$(2),PC%(2)
1540 IF EOF(1) THEN 1560
1550 N%=3:INPUT#1,N$(3),C$(3),A1$(3),
     A2$(3),PC%(3)
1560 LPRINT TAB(4);:FOR I%=1 TO N%
1570   LPRINT C$(I%);
1580   IF C$(I%)<>"" THEN PRINT" ";
1590   LPRINT N$(I%);TAB(I%*26+4);
1600 NEXT I%:LPRINT
1610 LPRINT TAB(4);:FOR I%=1 TO N%
1620   LPRINT A1$(I%);TAB(I%*26+4);
1630 NEXT I%:LPRINT
1640 LPRINT TAB(4);:FOR I%=1 TO N%
1650   LPRINT A2$(I%);",";PC%(I%);
       TAB(I%*26+4);
```



```
1660 NEXT I%:LPRINT:LPRINT:LPRINT
1670 IF EOF(1) THEN 1690
1680 GOTO 1500
1690 CLOSE 1:LPRINT CHR$(12)
1700 GOTO 220
1710 :
1720 REM ** SUBROUTINE TO LIST TO CON: **
1730 :
1740 PRINT CHR$(12):PRINT:PRINT:
     PRINT TAB(30);"File Listing:"
1750 WIDTH 80
1760 OPEN "I",1,F$
1770 IF EOF(1) THEN 1790
1780 N%=1:INPUT#1,N$(1),C$(1),A1$(1),
     A2$(1),PC%(1)
1790 IF EOF(1) THEN N$(2)="":C$(2)="":
   A1$(2)="":A2$(2)="":PC%(2)=0:GOTO 1810
1800 N%=2:INPUT#1,N$(2),C$(2),A1$(2),
     A2$(2),PC%(2)
1810 IF EOF(1) THEN N$(3)="":C$(3)="":
   A1$(3)="":A2$(3)="":PC%(3)=0:GOTO 1830
1820 N%=3:INPUT#1,N$(3),C$(3),A1$(3),
     A2$(3),PC%(3)
1830 PRINT C$(1);
1840 IF C$(1)<>"" THEN PRINT " ";
1850 PRINT N$(1);TAB(26);C$(2);
1860 IF C$(2)<>"" THEN PRINT " ";
1870 PRINT N$(2);TAB(52);C$(3);
1880 IF C$(3)<>"" THEN PRINT " ";
1890 PRINT N$(3)
1900 PRINT A1$(1);TAB(26);A1$(2);
     TAB(52);A1$(3)
1910 PRINT A2$(1);",";PC%(1);TAB(26);
     A2$(2);
1920 PRINT ",";PC%(2);TAB(52);
     A2$(3);",";PC%(3)
1930 PRINT:PRINT
1940 IF EOF(1) THEN 1960
1950 GOTO 1770
1960 CLOSE 1:WIDTH 72:INPUT "Hit return
     to continue";A$
1970 GOTO 220
1980 END
```

The structure of the complete program reflects some of the weaknesses of the BASIC language. In particular, the two subroutines at lines 1250 and 1360 are in the middle of the program.

For absolute speed (not important here) they should be near the beginning of the program, while for logical structure they should be near the end. However, once they have line numbers they can't be moved and new modules (in this case the label printer and 'list to console' module) have to be tacked on after them.

That completes our first major project. Now we'll move on to something new.

□

Notes

Well, after that tour de force of listings (did you make it through the last chapter?) Les continues his tutorial saga with a snappy little piece about random access files . . .

PART VI

WE'VE NOW seen how sequential files can be used to store information and how the computer can sort information into order.

These concepts are very important in organising information on a computer, particularly for rapid access later. But you have probably spotted the drawbacks of sequential files. First, we have to load the entire file into memory to sort it, and second, the only way to find a particular record is to sequentially search through the file until it's located.

Random access files will help us to get around these problems. The solution to the second problem is implicit: instead of sequentially searching a file, we can randomly access any part of it. Our earlier use of an index will help us with the first objection: instead of loading the entire file into memory and sorting it, why not have an index file which says where in the main file each record is stored, and then sort the index?

Of course there is a penalty for random access. We must always use the same length of record, and we must be a little bit more pernickety in defining our field lengths. But that's a small price to pay.

With the solution to our problems in sight, let's press on and learn about random access files.

A random access file is opened for both reading and writing, using our old friend the OPEN statement:


```
170 OPEN "R", 1, "FILENAME.DAT"
```

Now from here on in it's not going to be that easy, I'm afraid! To use random access files you must understand a little about how BASIC accesses the disk and how the operating system organises the disk.

Keeping Track Of Floppy

For convenience I will use the IBM standard single-density 20 cm floppy disk as our example, but the basic principles apply to all disks, whatever their size. Each disk has 77 **tracks**. These are concentric rings where the read/write head actually puts the data. Each track is split up into 26 **sectors**, each containing 128 bytes of data. Now we can access any block of data, provided we know which track and sector it's in.

A special area of the disk is usually set aside for the **directory**. This tells the operating system where each file starts and how long it is.

In random access files, each read or write (GET or PUT statement) transfers an entire sector of 128 bytes. (If your machine uses 256-byte sectors, don't worry, the operating system automatically *deblocks* a 256-byte **physical sector** into two 128-byte **logical sectors**).

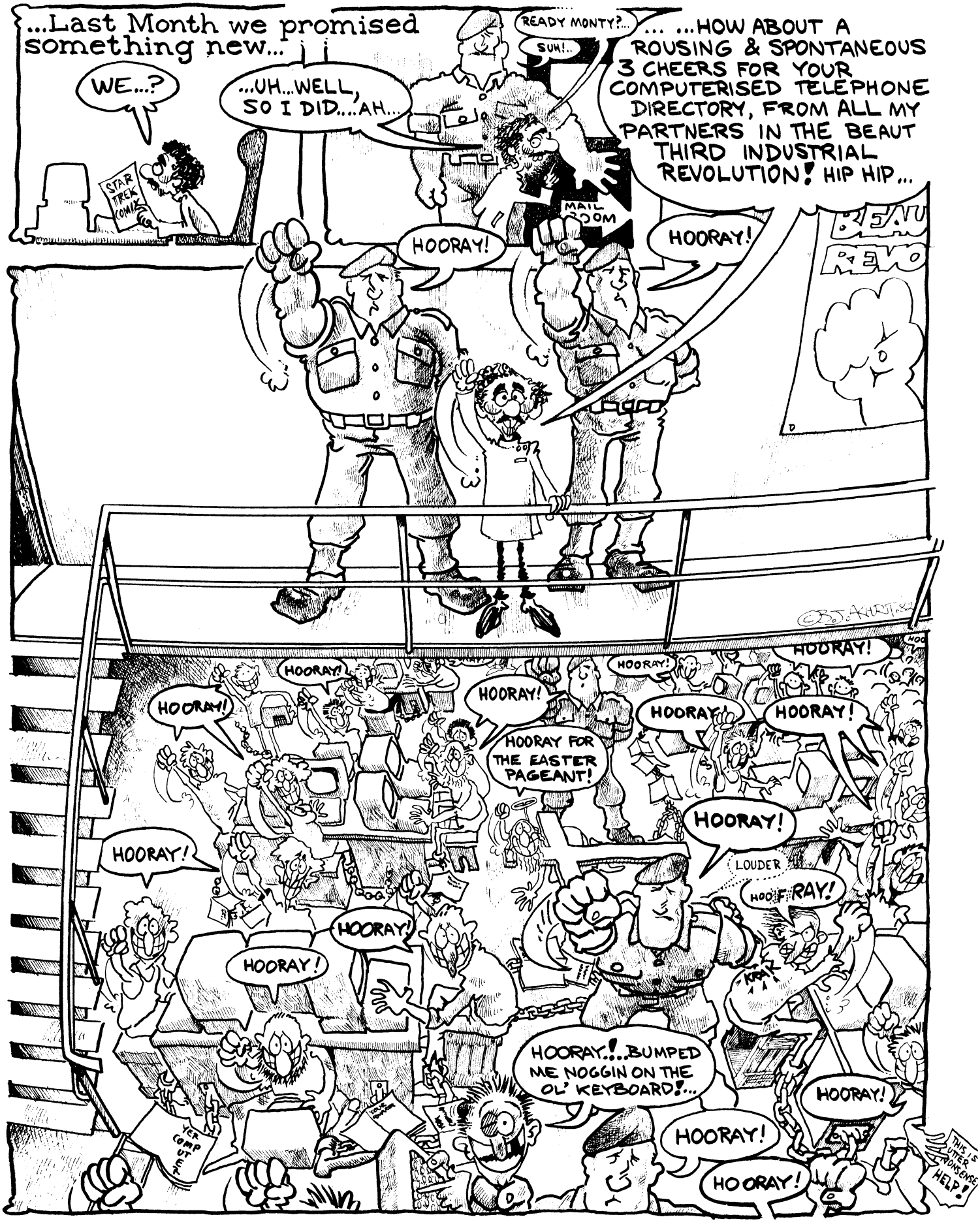
The sector is read off the disk into a buffer space maintained by BASIC, and which you should have already set up using a FIELD statement.

All data in a random access file is treated as string data. Thus any numbers in the file have to be converted into numeric data before being used in calculations. BASIC contains special functions to do this. Conversely, numbers must be converted into strings before being written to random files.

When storing strings into an I/O buffer, you must use the special functions LSET and RSET. This is because the fields are fixed in length and these functions automatically cope with strings that are too short or too long.

How Many Files, Sir?

Most BASICs allow 15 or so files to be


...Last Month we promised something new...
WE...?
STAR TREK COMIC
...UH...WELL, SO I DID....AH...
READY MONTY?...
SUH!..
... ...HOW ABOUT A ROUSING & SPONTANEOUS 3 CHEERS FOR YOUR COMPUTERISED TELEPHONE DIRECTORY, FROM ALL MY PARTNERS IN THE BEAUT THIRD INDUSTRIAL REVOLUTION! HIP HIP...
MAIL ROOM
HOORAY!
HOORAY!
BEAU REVO
HOORAY!
HOORAY!
HOORAY!
HOORAY!
HOORAY!
HOORAY!
HOORAY!
HOORAY FOR THE EASTER PAGEANT!
HOORAY!
HOORAY!
HOORAY!
HOORAY!
LOUDER
HOORAY!
KRAK
HOORAY!.. BUMPED ME NOGGIN ON THE OL' KEYBOARD!...
HOORAY!
HOORAY!
HOORAY!
HOORAY!
TER COMPUTER
THIS IS UTTER NONSENSE
HELP!

open at one time. TRS-80 disk BASIC asks the user, at power-up, how many files he will be using, because each file has to have an associated buffer in memory: if you're not using the buffer, it's just so much wasted space. Not many programs would have as many as 15 files open at once.

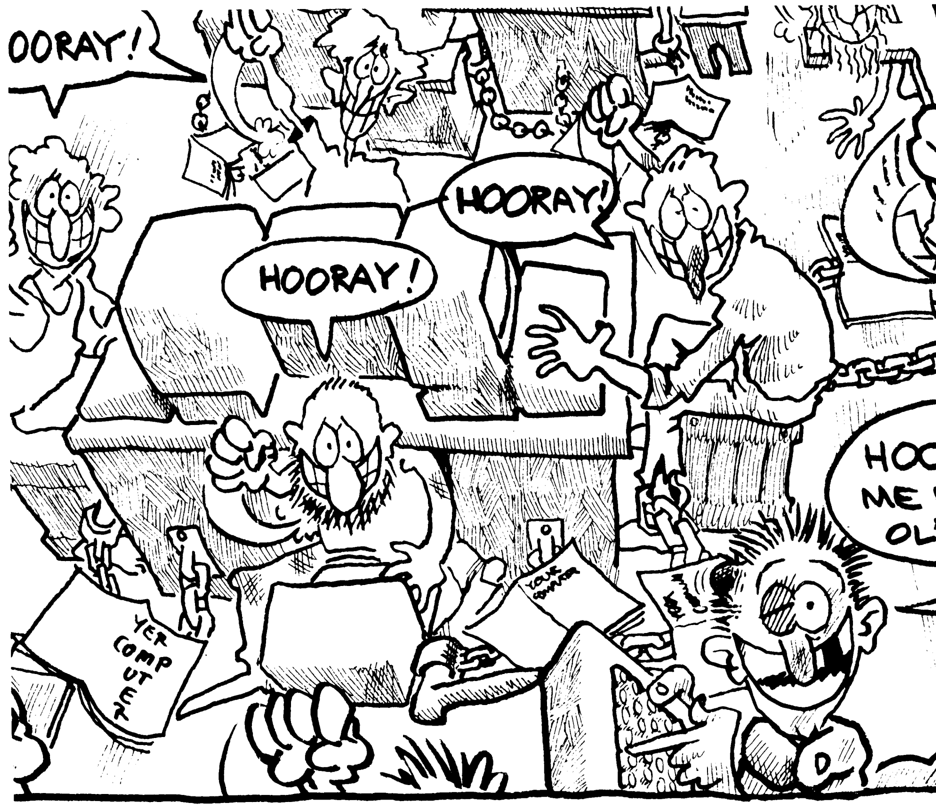
The OPEN statement sets up a 128-byte buffer for that particular file. The next thing you must do is say what information goes where inside that buffer. You should decide this during the early stages of designing your program, by writing a **record definition**.

Here's a typical record definition for a name/address/telephone number file:

Surname 20 bytes (characters)
Christian names 20 bytes
Street address 30 bytes
Town/city 20 bytes
Postcode 4 bytes
Telephone 15 bytes
Comments 19 bytes

Total length 128 bytes

If you like, you can visualise the buffer as a strip of memory containing the various strings which will represent the different fields of the record:




```
$$$$$$$$$$$$$$$$$$$$)($$$$$$$$$$$$$$$$$$$$)($$$$$$$$$$$$$$$$$$$$$$$$$$$$$$)
     SURNAME                 CNAME                     STREET ADDR

($$$$$$$$$$$$$$$$$$$$)($$$$)($$$$$$$$$$$$$$$)($$$$$$$$$$$$$$$$$$$)
     TOWN/CITY          P/C       TEL              COMMENTS
```

Fields Within A Buffer

Using the same string variable names as the mailing list program, here's a FIELD statement to set up the buffer this way:

```
180 FIELD #1,20 AS N$, 20 AS C$, 30 AS
    A1$, 20 AS A2$, 4 AS PC$, 15 AS T$,
    19 AS CM$
```

This statement 'slices up' the buffer for file 1 so the appropriate number of characters is allocated for each string variable.

Note: we have now reserved these string variables for a special usage and we cannot use them in the ordinary ways we use other string variables. For that reason, it is generally wise to use special names for disk-buffer variables, so the ubiquitous N$ is used in the program generally, while NF$ is the file buffer variable (FN$ cannot be used, as we'll see later).

So it might be better to write:

```
180 FIELD #1,20 AS NF$, 20 AS CF$, 30 AS
    AF$, 20 AS BF$, 4 AS PF$, 15 AS TF$,
    19 AS DF$
```

Although these string names are not as meaningful as the others, they are less likely to be used accidentally as conventional variables.

Okay, so how do we read from a random file? Having opened the file and FIELDed it, here's how to read the 37th record:

```
330 GET #1,37
```

That will read the record into the buffer. You can now PRINT NF$, and the name should appear on the screen. Likewise PRINT TF$ should print the phone number.

All of this assumes there is 37th record of course. If you don't have many friends you may not have a record number 37. In that case, what you get back will either be garbage, or possibly an error message.

One of the most convenient features of a random file is not every record need contain any information. In fact, there can be thumping great gaps in your file. Again, if you access a non-existent record, you'll generally get back garbage.

The disadvantage of this arrangement is non-existent records still take up space. If you create a random file, and then insert record number 1 followed by record number 1017, the file will occupy 1017 by 128 bytes — totalling 127 Kbytes — even though the space between records 1 and 1017 is logically empty. Be aware of this problem!

So You're Mismatched, Eh?

How do you know where the end of a random file is? The answer is random files don't really have a length — they just end somewhere after the last record. Consequently, the EOF() function doesn't work on random files; trying to take the EOF() of a random file will usually return a 'File Mismatch Error'.

To help with applications where you want to read right through a file, MBASIC provides a function, LOF(n), which returns the number of records in random file n. So you can write:

```
140 FOR N=1 TO LOF(2)
150 GET #2,N
160 REM DO PROCESSING
 .
 .
270 NEXT N
```

This will read through the file and process the contents of every record. Don't be surprised if many of them turn out to be garbage!

Numeric values are not quite as easy to handle as strings, however. In fact in random files, numeric values must be stored in a compressed string format. They are converted using the MKI$, MKS$ and MKD$ functions.

When using FIELD to set up a buffer, you must be sure to allow for the correct number of bytes (characters) for the data type:

Integer : string MKI$ 2 bytes
Single-precision : string MKS$ 4 bytes
Double-precision : string MKD$ 8 bytes

When reading numeric variables back from a random file, they must be converted back into numeric form. Once again, MBASIC provides special functions to accomplish this:

String : integer CVI
String : single-precision CVS
String : double-precision CVD

Those are the basic principles of handling random files, for simple cases at least. With this information we will proceed to a couple of more realistic examples: a computerised telephone directory and a small database management system. □

Notes

By now you must be an avid bird-watcher. Believe us, these programming skills will be a feather in your computing cap. In this final episode of the first half of our tutorial (got that?) Les has written a gentle introduction to forms design. You'll meet such interesting fauna as terminal functions, user-definable functions and the dummy variable (whoever he might be).

PART VII

I'LL BEGIN this month by temporarily diverting your attention from disk files to the point where the whole process starts — with the information being input — and show you how inputting can be made neater and more professional.

To do this we're going to tackle screen handling and user-definable functions.

The BASICs in some machines include simple screen-handling functions, in the form of the CLS and PRINT@ statements, so certain parts of this information will not apply to users of TRS-80, System 80 and similar systems. However, some of the other techniques may prove useful.

Inputting data to a computer is like filling in a form — the whole process can be made far more comfortable if the screen is organised to look just like a form. To do this with a standard 'dumb' computer terminal we must output special control codes or escape sequences to the terminal. These codes command it to perform such functions as clear screen, position cursor, and so on.

Different terminals have different sequences of control codes for their functions; that's why a program like WordStar, which makes extensive use of terminal functions, usually comes with an INSTALL program. INSTALL sets up the appropriate codes in the program to make it work on a particular terminal. You can do something similar in your programs by setting up the appropriate codes in string variables and functions at the beginning of a program.

For example, the Lear-Siegler ADM-3A terminal will clear its screen if you send a CHR$(26). In the case of a Televideo TVI 910, the appropriate character sequence is CHR$(27) followed by CHR$(26). For an ADDS Regent terminal, it is CHR$(12).

Times Getting Tougher Than...

Okay, imagine someone gives you a program written for the ADDS with PRINT CHR$(12) statements all the way through it, and you own a TVI 910.

You're in luck if you have a text editor with global search and replace function: otherwise, you'll have to go through and find every occurence of PRINT CHR$(12) and change it to PRINT CHR$(27);CHR$-(26). You'll also have to watch out for LPRINT CHR$(12) statements — they send form feeds to the printer...

Life was meant to be easier than this, wasn't it?

Surely it would have been better if your friend had written at the beginning of the program:

```
10 CLS$ = CHR$(12)
```

and then used PRINT CLS$ all the way through. All you would have to do to be in business is change line 10 to:

```
10 CLS$ = CHR$(27) + CHR$(26)
```

As a bonus this even works faster, though you're unlikely to notice the difference.

You can use a similar technique to send cursor home, up, down, left and right commands to any terminal. For example, for the TVI 910:

```
10 HM$ = CHR$(30)
20 CLS$ = CHR$(27) + CHR$(26)
30 UP$ = CHR$(11)
40 DN$ = CHR$(10)
50 LE$ = CHR$(8)
60 RT$ = CHR$(12)
```

Now we're starting to get somewhere! But how about more complex jobs, like moving the cursor to a particular row and column? For the ADM-3A this means sending an escape character, CHR$(27), followed by an equals sign, then the row number plus an offset of 31 (as a binary number), then the column in the same fashion.

How do we treat this case?

That's where **user-definable functions** come in. In Microsoft BASIC and CBASIC-2 you can define your own functions by using the DEF FN statement. For example, we can define a function which converts Centigrade to Fahrenheit. The formula is F = 1.8 by C + 32 (we used it earlier in a simple program).

To define a function called FNF(C) we write:

```
70 DEF FNF(C) = 1.8 * C + 32
```

In this case, the function is really called F, or FNF (function F) in full. The bracketed C is a **dummy variable**. It bears no relation to the variable C which may be used elsewhere in a program. It simply means the number appearing in brackets when the function is called


...Last issue we were slightly irrelevant.... This month we're back on the wrong track...
I DEMAND AN AUDIENCE WITH L.B. TO DISCUSS THE RELEVANCE OF MY PRESENT CONDITION TO MY DISCHARGE FROM 'SUNNY BELL' HOME FOR BURNT OUT HACKERS...
...RELEVANCE HAS NUFFIN TO DO WIVVIT SON...
TAP TAP TAP
SNAP!
ELEKTRIC SHOCK!
..ANYHOW...AH HAH! A USER DEFINABLE FUNCTION...
PARDON?
THROB THROB
FINGER POINTING...IN FACT YOU MIGHT SAY IT'S A TECHNIQUE TO PRODUCE RUDIMENTARY GRAPHICS
I WANNA LEAVE!
OH
E.G. OF RUDI-MENTARY GRAPHICS
...YOU'RE FREE TO GO RIGHT NOW!.. IT'S BEEN A GREAT PLEASURE HELPING YOU!
UNDO HIS LEG IRONS
...P..PLEASE...THERE IS JUST ONE THING I WANT OF YOU... ...BEFORE YOU GO...
...UH...WHA...
...WHAT'S THAT...?..
...TIDY YOURSELF UP A BIT... TUCK IN YOUR SHIRT...DO UP THAT SHOE LACE...COMB YOUR HAIR...HERE, USE MY OL' NYLON COMB...
WHAT?
....OH! OH.. SURE!... GEE, THANKS OL' MATE..
ZZT!
...NOW, TAKE HIM BACK AND RE-PROGRAM HIM...
...NOW YOU ALL KNOW WHY THE BEST HACKERS ARE ALSO THE SCRUFFIEST... A.R.S.E.
AVOID RAISING STATIC ELECTRICITY..
...UM.. AND NOW, BACK TO THE RELEVANT ISSUE...
...I'M JUST TRYING YOUR NEW UNIFORM ON THE DUMMY VARIABLE, MR BELL SIR...
BIGGER
SMALLER
TALLER
SHORTER
©B.J. AKIRT. '82

should be used as the dummy variable C in the calculation.

Use Of The Dummy

To show the use of this simple function in a program, here's an example:

```
10 DEF FNF(C) = 1.8 * C + 32
20 INPUT "Centigrade";X
30 PRINT "equals";FNF(X);"Fahrenheit"
40 END
```

Note the function must be defined before it is used, and that although in the definition C is used as the dummy variable, when it is called will operate on whatever variable is passed to it.

Now in this example the function returned a real number. However, functions can return other data types too — and function names follow the same rules as variable names. So an integer function could be defined and named FNDO%(X), or a string function named FNHS$(A$).

To return to our problem of PRINTing the escape sequence which positions the cursor, here's a possible solution:

```
10 DEF FNGXY$(X,Y)=CHR$(27)+"="
   +CHR$(Y+31)+CHR$(X+31)
```

Here we've defined a function GXY$ (gotoXY) which is to be passed two dummy variables X and Y. It then constructs a string consisting of ESCape, =, Y plus offset, X plus offset. On an ADM-3A, then, the line

```
240 PRINT FNGXY$(40,12);
```

will position the cursor near the centre of the screen.

Different functions will be required for other terminals; on a Hazeltine terminal you would use this function:

```
10 DEF FNGXY$(X,Y) = CHR$(27)+CHR$(17)
   +CHR$(X+31)+CHR$(Y+31)
```

On the ADDS Regent, it would be

```
10 DEF FNGXY$(X,Y) = CHR$(27)+"Y"
   +CHR$(Y+63)+CHR$(X+63)
```

This technique can be used to produce rudimentary computer graphics on serial terminals. Try this short program — after modifying it for your terminal, of course:

```
100 DEF FNGXY$(X,Y)=CHR$(27)+"="
    +CHR$(Y+31)+CHR$(X+31)
110 CLS$=CHR$(27)+CHR$(26)
120 AS$ = CHR$(27)+ CHR$(72)
130 PRINT AS$
140 PRINT CLS$
150 FOR X=1 TO 80
160   FOR Y=1 TO 24
170     PRINT FNGXY$(X,Y);"*";
180   NEXT Y
190 NEXT X
200 PRINT AS$
```

The AS$ is a string to turn auto-scrolling on and off on the TVI 910. Without it the terminal will automatically scroll up periodically, spoiling the display.

Well It's Round-ish, Sir

To display a circle (of sorts) try this:

```
100 DEF FNGXY$(X,Y)=CHR$(27)+"="
    +CHR$(Y+31)+CHR$(X+31)
110 CLS$=CHR$(27)+CHR$(26)
120 AS$ = CHR$(27)+ CHR$(72)
130 PRINT AS$
140 PRINT CLS$
150 FOR X = -10 TO 10
160   YC1 = 12 + SQR (100 _ X ^ 2)
165   YC2 = 12 _ SQR(100 _ X ^ 2)
170   XC = 60 + X
180   PRINT FNGXY$(XC,YC1);"*";
      FNGXY$(XC,YC2);"*";
190 NEXT X
200 PRINT AS$
210 PRINT FNGXY$(1,1);:LIST
```

Many terminals have other functions. On the TVI 910, for example, the strings 'ESC)' and 'ESC (' turn half-intensity on and off. So we can define a function to print a string in half intensity:

```
30 DEF FNHT$(A$)=CHR$(27)+")"+A$
   +CHR$(27)+"("
```

To print a string of underlines:

```
40 DEF FNLN$(L)=STRING$(L,95)
```

Here the STRING$ function is used to generate a string of length L, composed of underlines (CHR$(95)). To print a string, but underlined (on the TVI 910):

```
50 DEF FNUL$(A$)=CHR$(27)+"G8"+A$
   +CHR$(27)+"G0"
```

Take a look through the manual for your terminal; you will find many functions that can be controlled this way. It's possible to lock and unlock the keyboard, turn a printer on and off, make characters blink or inverse video, and so on.

The most important use of these functions is in the creation of forms for input and formatted output. In the case of our telephone directory, we are dealing with fixed length records, which cannot be exceeded. It would be handy to know how much space is available for a name before we start filling it in. We can do this by printing up a blank form with underlines indicating the space available for data entry.

To take us out this month, here's an example of a short routine which could be used in our telephone directory program.

```
10 DEF FNGXY$(X,Y)=CHR$(27)+"="+CHR$(Y+31)
   +CHR$(X+31)
12 DEF FNHT$(A$)=CHR$(27)+")"+A$
   +CHR$(27)+"("
13 DEF FNLN$(L)=STRING$(L,95)
14 HOME$=CHR$(30)
15 CLS$=CHR$(27)+CHR$(26)
17 PRINT CLS$
20 PRINT FNGXY$(1,3);FNHT$("Surname     : ");
   FNLN$(20)
30 PRINT FNHT$("First Name  : ");FNLN$(20)
40 PRINT FNHT$("Street      : ");FNLN$(30)
50 PRINT FNHT$("Town/City   : ");FNLN$(20)
60 PRINT FNHT$("Postcode    : ");FNLN$(4)
70 PRINT FNHT$("Telephone   : ");FNLN$(15)
75 PRINT FNHT$("Comment     : ");FNLN$(19)
80 PRINT FNGXY$(13,3);:INPUT N$
90 PRINT FNGXY$(13,3);": ";N$;
   SPACE$(20-LEN(N$))
100 PRINT FNGXY$(13,4);:INPUT C$
110 PRINT FNGXY$(13,4);": ";C$;
    SPACE$(20-LEN(C$))
120 PRINT FNGXY$(13,5);:INPUT A1$
130 PRINT FNGXY$(13,5);": ";A1$;
    SPACE$(30-LEN(A1$))
140 PRINT FNGXY$(13,6);:INPUT A2$
150 PRINT FNGXY$(13,6);": ";A2$;
    SPACE$(20-LEN(A2$))
160 PRINT FNGXY$(13,7);:INPUT PC%
170 PRINT FNGXY$(13,7);":";PC%;SPACE$(16)
180 PRINT FNGXY$(13,8);:INPUT TEL$
190 PRINT FNGXY$(13,8);": ";TEL$;
    SPACE$(15-LEN(TEL$))
200 PRINT FNGXY$(13,9);:INPUT CT$
210 PRINT FNGXY$(13,9);": ";CT$;
    SPACE$(19-LEN(CT$))
```